A TEST-BED FOR LINEAR TIME, CONSTANT SPACE DIFFERENCING ALGORITHMS

By

Pauline Mouawad

A Thesis

Submitted in Partial Fulfillment of the Requirements for the

Degree of Master of Science

in Computer Science

Department of Computer Science

Faculty of Natural and Applied Sciences

Notre Dame University – Louaize

Zouk Mosbeh, Lebanon

June 2004

A Test-Bed for Linear Time, Constant Space Differencing Algorithms

By

Pauline Mouawad

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of Master of

Science in Computer Science

Department of Computer Science

Faculty of Natural and Applied Sciences

Notre Dame University – Louaize

Zouk Mosbeh, Lebanon

June 2004

A Test-Bed for Linear Time Constant Space Differencing Algorithms

By

Pauline Mouawad

Approved:

Fouad Chedid

Fouad Chedid: Professor of Computer Science

Advisor & Chairman of Committee

[illegible]

Issam Moghrabi: Associate Professor of Computer Science

[illegible]

Omar Rifi: Assistant Professor of Computer Science

[illegible]

Ziad Rached: Assistant Professor of Mathematics

Date of Thesis Defense: June 22nd, 2004

ACKNOWLEDGEMENTS

"Let us, then, be up and doing, with a heart for any fate; still achieving, still pursuing, learn to labor and to wait."

Henry Wadsworth Longfellow

If one finds passion in one's work, how can this merriment be full if it has no gratitude to show? I cannot be content with a work that is merely the fruit of a singular root but rather, I draw satisfaction from the knowledge that ultimately, I have someone to thank.

My thanks are due to my parents and my sister, for always being there for me. Their encouragement and support have been most precious to the fulfillment of this achievement. I couldn't have done this without them.

To my advisor, Dr. Fouad Chedid, I offer heartfelt thanks for the pillar he has been for this work. His tireless dedication and truthful support have been an invaluable guidance to me and an incitement to my resolution. It is said that: "*The teacher who is indeed wise does not bid you to enter the house of his wisdom but rather leads you to the threshold of your mind*" (Khalil Gibran). I could not have said this any better.

My deepest respect and gratitude to Notre Dame University, that gave me the education and knowledge I sought. I also owe a well of gratitude to the teachers who have taught me throughout the graduate study; they truly made this possible. I dedicate this thesis to my family and loyal friends; they really make the world a nicer place to live in.

ABSTRACT

This thesis is motivated by the latest work related to Differential compression algorithms as it appears in the work of Ajtai, et al. - 2002. In particular, we pay special attention to delta encoding algorithms that achieve good compression in linear time and constant space. This is important because previous work in this area uses either quadratic time and constant space or linear time and linear space, which is unacceptable for large inputs. In delta encoding, the algorithm reads two different copies of the same file as input, termed the reference copy and the version copy. The output of the algorithm is a sequence of Add/Copy commands that reconstructs the version copy in the presence of the reference copy. Such algorithms have been recommended to be integrated into the http protocol. The idea is to reduce the data transfer time for text and http objects in order to decrease the latency of loading updated web pages. Also, in a client server system, clients may perform delta encoding to exchange delta encodings with a server instead of exchanging whole files. This reduces the time needed to perform the backup and reduces the storage required at the backup server. In the literature, the evaluation of delta encoding algorithms depends on three metrics: the running time of the algorithm, the space it uses, and the compression results it achieves. In this thesis we build a test-bed for delta encoding algorithms that accommodates most of the previous work described in the literature. Through intensive experimentations we are able to recommend a hybrid algorithm that forms a good compromise among the existing methods.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER I

The Problem

During the last decade, the field of computer science has witnessed a widespread development that grew significantly. This growth triggered an increase in the demand for fast data transmission over the network (Internet or intranet). The solution was to compress large files into smaller versions so that they can be managed easily. Various techniques have been suggested and consequently provided solutions to the problem. Unfortunately limitations were encountered because no matter what the technique is, compressed files are still too large for a remote transmission and still require enormous transfer time and create unacceptable traffic over the network. Concisely, time and space are still a major issue to be dealt with.

Recent studies have used differential compression techniques in an attempt to solve the problem and various algorithms have been proposed [1, 3, 4, 6, 7, 8, 9]. The main concept is to try to find common strings between two versions of data and use them to encode one version compactly; this is accomplished by describing the encoding itself as a set of changes of one version with respect to its companion. Instead of compressing a whole new file to replace wholly an older version of it, we would rather check for the changes in it and generate some code that would only send those changes to the older version. A simple process ultimately combines the file with the changes. This is quite an interesting approach because it considerably cuts the size of the file to be compressed.

Recently, new algorithms have been put that run in linear time and use constant space. But the tradeoff is with the compression results. The algorithm has to disregard some strings in the file in favor of its running time and space usage. And sometimes a significant portion in the file is not encoded because the algorithms favor the recent scanned strings over the previously read ones and therefore information about past strings is completely lost. More on this will be described in the next chapter.

This thesis introduces a new algorithm, which attempts to yield near to optimal results by checking almost every string of length *p* in both files: the old one called *Reference* and the new one called *Version*. Similarities found are encoded with a *Copy* command from the *Reference* while non-matching strings are encoded with an *Add* command from the *Version*. The method used ensures a *window into the past* of the files being scanned in a sense that it examines almost all strings and it does not favor a string over the other except for strings that yield longer matches. We keep in mind that one string may match differently depending on how long it can extend in the file and still matches its copy in the other file. For example, a string of length *p = 3* consisting of *abc* in the *Reference* file that only matches to *abc* in the *Version* file is far less beneficial than the same string *abc* that extends in both the *Reference* and *Version* files to become *abcdefghijklmnop* of length *p = 16*. Therefore this considerably diminishes the amount of strings left un-encoded. Moreover, instead of having several *copy* commands to encode a string of length p = 16, *1 copy* command does so and clearly decreases the cost of the compression. Also and very importantly, this algorithm cuts down on transmission time and on storage space. Ultimately, only variations of the file are being transmitted to the user, which means a major cut down

on Internet transfer time to the user's satisfaction. Finally and at the user-end, the program is run and the file is assembled and becomes the new updated version.
The fashion by which the cost of such technique is computed will be described in a later chapter, in full.

The rest of the thesis is organized as follows. Chapter II offers an overview of related algorithms from the literature. Chapter III is our contribution in which we describe a *Test-Bed* of differencing algorithms that run in linear time and constant space. The actual code is included in the Appendix and heavily commented for a clear and friendly reading. Chapter IV lays down a series of tests conducted on different files of different sizes while varying the size of a specific variable L in the code to be revealed later on. A series of Table and Graph representations will accompany those tests for a concise and obvious conclusion about the Average Cost of our algorithm as well as the time and space complexities. Chapter V is the conclusion.

CHAPTER II

Literature Review

Given two strings V and R, differential compression is about encoding *V* with respect to *R* by finding regions of *V* identical to regions of *R* and encoding this data with a reference to the location and size of the data in *R* [1]. At each step during execution, the algorithm examines strings, also called seeds, having a constant length *p*. Generally, the inputs to a differential compression algorithm are strings from both the *Reference* and *Version* files, and the output is a *delta string*:

$$\Delta(\mathbf{R}: \mathbf{V}) = A(\mathbf{R}: \mathbf{V})$$

The output Δ will consist of a sequence of commands that will be described in the next section under *Delta Encoding*.

As mentioned earlier, the performance of this algorithm is measured using three metrics:

1. The time complexity. It is the running time of the algorithm.
2. The space complexity. It is the storage space the algorithm uses in order to process the strings read from both files.
3. The compression results achieved. These are the series of commands issued that differentiate between the similarities and the differences among the two versions of a file. Hence the term differential compression.

The purpose of this type of encoding is to find algorithms that would ultimately, no matter the file sizes or the scale of the input, generate a near to optimal compression result while cutting down on transmission time and storage space.

In this study, all the algorithms discussed follow the *Delta Encoding Compression Algorithm* which we describe next.

2.1 Delta Encoding Compression Technique

Given two copies of a file, the *Reference* - standing for the old version - and the *Version* - being the new version - , the delta encoding algorithm is about finding the longest match possible of a given string after performing a left-to-right scan of both files in a parallel fashion in search of matching seeds of an initial length *p*. In the event where such match is found, an attempt to *extend* that match to a longer one takes place. This is called the *extension phase.*

When a matching string is located, a command is issued in the form:

$$(C, l, a)$$

The character C stands for the command *Copy*, *l* stands for the length of the string to be copied, and *a* stands for the address. This reads into the following: Copy a string of length *l* found at offset *a* in the *Reference* file.

On the other hand, when a non-matching string is encountered, a different command is issued in the form:

$$(A, l, S)$$

The character A represents the command *Add*, *l* stands for the length of the string to be added, and *S* is the actual string being added. And it reads: Add a string *S* of length *l* from the *Version* file.

A simple example follows to illustrate this concept:

Given: $R = A\,B\,C\,D\,E\,L\,M\,N\,O\,P\,Q\,R\,S\,T\,X\,Y\,Z$

&

$V = F\,G\,H\,I\,J\,K\,L\,M\,N\,O\,P\,U\,V\,W\,X\,Y\,Z$

A Delta Encoding output would be:

[(A, 4, 'FGHIJK'), (C, 5, '5'), (A, 3, 'UVW'), (C, 3, '14')]

The identification of fixed length matching strings is done by reducing the seed to a fixed integer by means of a *hash* function *F*. The resulting integer of the form *F(S)* is called the seed's *footprint.* Assuming an ideal hashing function, one footprint should uniquely identify a given seed; but such consideration would be misleading since in a real life application, two different seeds might in fact hash to the same footprint value.

The hashing function *F* generates a hash table with the number of its entries equal to the number of footprints generated. Indeed, a hash table entry is a footprint and a hash table size depends on the number of footprints stored. A hash table entry that corresponds to one or more seeds scanned may hold one or more offsets indicating the respective addresses of those seeds, depending on the algorithm used. Some algorithms do not allow more than one offset to be stored at a given footprint, as we shall see in the coming sections.

2.2 Notations

The notations used for differential compression are:

2.2.1 - Pointers:

The following pointers are defined in the code:

v_c: Current address of the *Version* string

r_c: Current address of the *Reference* string

v_s: Starting offset of a version string. This pointer holds the starting offset of the un-encoded suffix of the version string.

v_m: Offset of a matching version string.

r_m: Offset of a matching reference string.

2.2.2 – Parameters

The following parameters are defined as well:

p: Seed length. It's the length of substrings of which a footprint is calculated.

q: The size of the hash table = the number of footprint values.

$|X|$: Length of a string X

$n = |R| + |V|$: The combined length of the reference and version strings.

2.2.3 – Hashing function

The following hashing function due to Karp and Rabin [1987] is used in the code:

If $x_0, x_1, \ldots, x_{n-1}$ are the symbols of a string *X* of length *n*, let X_r denote the substring of length *p* starting at offset *r*. Thus,

$X_r = x_r x_{r+1} \ldots x_{r+p-1}$.

Identify the symbols with the integers 0, 1, ..., b – 1, where *b* is the number of symbols. Let *q* be a prime, the number of footprint values. To compute the modular hash value (footprint) of X_r, the substring X_r is viewed as a base-*b* integer, and this integer is reduced modulo *q* to obtain the footprint; that is:

$$F_X(a, a+p) = \left(\sum_{i=r}^{r+p-1} x_i b^{r+p-1-i} \right) \bmod q.$$

Using this method, a footprint function is specified by two parameters: *p*, the length of substrings (seeds) to which the function is applied; and *q*, the number of footprint values. The choice of *q* involves a trade-off between space requirements and the extent to which the footprint values of a large number of seeds have different footprints. Typically, a footprint value gives an index into a hash table. The advantage of this function is that footprinting allows an algorithm to detect matching seeds of length *p*, but the algorithms in the literature are most successful when these seeds are part of much longer matching substrings; in this case, a matching seed leads the algorithm to discover a much longer match. The hashing function notations are:

F_X (a, a + p): The footprint of a seed X, starting at offset '*a*' up to

offset $a + p$ where p is the substring's length.

H_X: The hash table, indexed by seeds' footprints.

H_X [i]: The *i*th element in hash table H_X. In general, hash table entries

contain the starting offsets of seeds indexed by footprint value.

2.3 Outline of the Algorithm

1. *Initialize the hash table(s).* Create empty hash table(s).
2. *Initialize pointers.* Set v_c, r_c and v_s to zero.
3. *Generate new footprints.* Generate a new footprint at v_c and at r_c if there is enough input string to the right of v_c and r_c to generate a new footprint. If at least one footprint was generated, continue at Step (4). If not, go to Step (8) to finish the encoding of *V* and terminate.
4. *Try to find a matching seed.* Use the newly generated footprint(s) and the hash table(s) to try to find a matching seed in *R* and *V*. In some algorithms, the new footprint(s) are also used to update the hash tables. If a matching seed is not found, increment v_c (and increment r_c) by one, and repeat Step (3). If a matching seed is found, continue at Step (5) to extend the match.
5. *Extend the match.* Attempt to extend the matching seed to a longer matching substring in both *R* and *V* by comparing symbols between *R* and *V*.
6. *Encode the match.* Encode the substring of *V* from v_s to the end of the matching substring by producing the appropriate command sequence; this will always end with a copy command that encodes the matching substring. Update v_s to the new start of the unencoded suffix.
7. *Update and return to top of main loop.* Update v_c (and r_c) and modify the hash tables if needed. Return to Step (3).

8. *Finish up.* If there is an unencoded suffix of *V*, encode this suffix with an add command.

2.3.1 – A Greedy Differencing Algorithm

This algorithm is based on [9]. The greedy algorithm first makes a pass over the reference string *R*; it computes footprints and stores in a hash table, for each footprint *f*, all offsets in *R* that have footprint *f* (colliding footprints are handled by chaining footprints at each value). The algorithm then moves the pointer v_c through *V*, and computes a footprint at each offset. At each step it does an exhaustive search, using the hash table and the strings *R* and *V*, to find the longest substring of *V* starting at v_c that matches a substring appearing somewhere in *R*. The longest matching substring is encoded as a copy command, v_c is set to the offset following the matching substring, and the process continues.

The pseudocode in Figure 2.1 outlines the major steps of the greedy algorithm. In Step(1), the algorithm hashes the contents of the reference string in a hash table where each entry is a footprint containing all the offsets having that footprint; in Steps(3) to (6), the algorithm then finds longest matching substrings in the version string and encodes them.

Obviously, the space used by this algorithm is dominated by the space for the hash table (= $|R| - p + 1$ offset values stored in linked lists). Since p is a constant, the space is proportional to $|R|$. Concerning the bound of the time complexity, at each offset in R, the algorithm spends $O(l)$ time, in the worst case, to find a matching substring of length at most l starting at this offset. Thus, the total time is $O(|V||R|)$, that is, O (n^2). It is known in [1] that the greedy algorithm provides a solution to the perfect differencing problem if $p \leq 2$. The pseudocode that follows represents the basic steps executed by the greedy differencing algorithm.

Given a reference string R and a version string V, generate a delta encoding of V as follows:

1. For all offsets in input string R in the interval [0, |R| - p], generate the footprints
2. Start string pointers v_c and v_s at offset zero in *V*.
3. If $v_c + p > |V|$ go to Step (8). Otherwise, generate a footprint $F_V(v_c, v_c + p)$ at v_c.
4. (and (5)) In this algorithm it is natural to combine the seed matching and substring extension steps into one step. Examine all entries in the linked list at $H_R[F_V(v_c, v_c + p)]$ (this list contains the offsets in *R* that have footprint $F_V(v_c, v_c + p)$) to find an offset r_m in *R* that maximizes *l*, where *l* is the length of the longest matching substring starting at r_m in *R and* at v_c in *V*. If no substring starting at the offsets listed in $H_R[F_V(v_c, v_c + p)]$ matches a substring starting at v_c, increment v_c by one and return to Step (3). Otherwise, set v_m and r_m to the start offsets of the longest matching substring found. (In this algorithm, $v_m = v_c$ at this point.) Let *l* be the length of this longest matching substring.
5. The longest match extension has already been done in the combined step above.

6. If $v_s < v_m$, encode the substring $V [v_s, v_m)$ using an add command containing the substring $V [v_s, v_m)$ to be added. Encode the substring $V [v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Set v_s to $v_m + l$.
7. Set v_c to $v_m + l$ and return to Step(3)
8. All of the remaining unencoded input has been processed with no matching substrings found. If $v_s < |V|$, encode the substring $V [v_s, |V|)$ with an add command. Terminate the algorithm.

Figure 2.1 Pseudocode for the greedy algorithm

2.3.2 – The One-Pass Algorithm

The one-pass differencing algorithm finds a delta encoding in linear time and constant space. It finds matching substrings in a *next match* sense. That is, after copy-encoding a matching substring, the algorithm looks for the next matching substring forward in both input strings and disregards the portion of R and V that precedes the end of the substring that has just been copy-encoded. Hence it has a linear time complexity. The one-pass algorithm does not store all offsets having a certain footprint; instead it stores, for each footprint, at most one offset in R and at most one in V. This makes the hash table for R smaller with a size q rather than |R| and more easily searched however the compression is not always optimal. This algorithm works in a *first-fit* fashion, which means that it retains only the first offset found after each flush of the hash table. Hence it uses O (q) space. The drawback though, is that in the presence of transposed data (with R as ...X ... Y and V as ... Y ... X ...), the algorithm will not detect both of the matching substrings X and Y. Pseudocode of the one-pass algorithm follows.

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) Create empty hash tables, H_V and H_R, for V and R. Initially, all entries are empty.

(2) Start pointers r_c, v_c, and v_s at offset zero. Pointer v_s marks the start of the suffix of V that has not been encoded.

(3) If $v_c + p > |V|$ and $r_c + p > |R|$ go to Step (8). Otherwise, generate footprint $F_V(v_c, v_c + p)$ when $v_c + p \leq |V|$ and footprint $F_R(r_c, r_c + p)$ when $r_c + p \leq |R|$.

(4) For footprints $F_V(v_c, v_c + p)$ and $F_R(r_c, r_c + p)$:

(a) Place the offset v_c (resp., r_c) into H_V (resp., H_R), provided that no previous entry exists. The hash tables are indexed by footprint. That is, if $H_V[F_V(v_c, v_c + p)]$ = empty assign v_c to $H_V[F_V(v_c, v_c + p)]$; similarly, if $H_R[F_R(r_c, r_c + p)]$ = empty, assign r_c to $H_R[F_R(r_c, r_c + p)]$.

(b) If there is a hash table entry at the footprint value in the other string's hash table, the algorithm has found a likely matching substring. For example, $H_V[F_R(r_c, r_c + p)] \neq$ empty indicates a likely match between the seed at offset r_c in R and the seed at offset $H_V[F_R(r_c, r_c + p)]$ in V. In this case set r_m to r_c and v_m to $H_V[F_R(r_c, r_c + p)]$ to the start offsets of the potential match. Check whether the seeds at offsets r_m and v_m are identical. If the seeds prove to be the same, matching substrings have been found. If this is the case, continue at Step (5) to extend the match (skipping the rest of Step (4b)). Symmetrically, if $H_R[F_V(v_c, v_c + p)] \neq$ empty, set v_m to v_c and r_m to $H_R[F_V(v_c, v_c + p)]$. If the seeds at offsets r_m and v_m are identical, continue at Step (5) to extend the match. At this point, no match starting at v_c or starting at r_c has been found. Increment both r_c and v_c by one, and continue hashing at Step (3).

(5) At this step, the algorithm has found a matching seed at offsets v_m and r_m. The algorithm matches symbols forward in both strings, starting at the matching seed, to find the longest matching substring starting at v_m and r_m. Let l be the length of this substring.

(6) If $v_s < v_m$, encode the substring $V\,[v_s, v_m)$ using an add command containing the substring $V\,[v_s, v_m)$ to be added. Encode the substring $V\,[v_m, v_m+ l)$ as a copy of the substring of length l starting at offset r_m in R. Set v_s to the offset following the end of the matching substring, that is, v_s to $v_m + l$.

(7) Set r_c and v_c to the offset following the end of the match in R and V, that is, set r_c to $r_m+ l$. Flush the hash tables by setting all entries to empty. We use a non-decreasing counter (version number) with each hash entry to invalidate hash entries logically. This effectively removes information about the strings previous to the new current offsets v_c and r_c. Return to hashing again at Step (3).

(8) All inputs have been processed. If $v_s < |V|$, output an add command for substring $V\,[v_s, |V|)$. Terminate the algorithm.

Figure 2.2 Pseudocode for the one-pass algorithm

2.3.3 – The Correcting One-Pass Algorithm

The correcting one-pass algorithm differs from the one-pass algorithm by:

(1) keeping all existing entries in the hash table tables after encoding a copy command; and

(2) discarding a prior offset that has a particular footprint in favor of the current offset having that footprint.

By retaining all entries in the hash table, the correcting one-pass retains information about past substrings i.e. retains a *window into the past* which enables it to detect nonsequential matching substrings, that is, substrings that occur in the version string in an order different from the order in which they occur in the reference string. The correcting one-pass algorithm extends matching strings both backwards and forwards. This ability of reverse matching permits the correction of early bad encodings and therefore an Add command of a string of length $p = 10$ with a cost equal to 10, can be corrected to one Copy command with a cost equal to 1 in the best case. Despite the similarity between the correcting one-pass algorithm and the one-pass algorithm, the correcting one-pass algorithm does not have the same linear running time guarantee. The algorithm spends a large amount of time extending matches backwards at many executions, so that the total time spent during backwards matching grows faster than linearly in the length of the input.

The correcting one-pass algorithm pseudocode is:

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) Create empty hash tables, H_V and H_R, for V and R.

(2) Start pointers r_c, v_c, and v_s at offset zero. Pointer v_s marks the start of the suffix of V that has not been encoded.

(3) If $v_c + p > |V|$ and $r_c + p > |R|$ go to Step (8). Otherwise, generate footprint F_V $(v_c, v_c + p)$ when $v_c + p \leq |V|$ and footprint F_R $(r_c, r_c + p)$ when $r_c + p \leq |R|$.

(4) For footprints F_V $(v_c, v_c + p)$ and F_R $(r_c, r_c + p)$:

(a) Place the offset v_c (resp., r_c) into H_V (resp., H_R). That is, assign H_V $[F_V(v_c, v_c + p)]$ to v_c and H_R $[F_R$ $(r_c, r_c + p)]$ to r_c.

(b) As in the one-pass algorithm, if H_V $[F_R$ $(r_c, r_c + p)] \neq$ empty, set r_m to r_c and v_m to H_V $[F_R$ $(r_c, r_c + p)]$ to the start offsets of the potential match. If the seeds at offsets r_m and v_m are identical, continue at Step (5) to extend the match (skipping the rest of Step (4b)). Symmetrically, if H_R $[F_V$ $(v_c, v_c + p)] \neq$ empty, set v_m to v_c and r_m to H_R $[F_V$ $(v_c, v_c + p)]$. If the seeds at offsets r_m and v_m are identical, continue at Step (5) to extend the match. At this point, no match starting at v_c or starting at r_c has been found. Increment both r_c and v_c by one, and continue hashing at Step (3).

(5) Having found a matching seed at Step(4b), extend this match forwards and backwards from v_m and r_m as long as possible, reset v_m and r_m to the start offsets of the matching substring (if the match extended backwards), and set l to the length of the matching substring.

(6) Encode the matching substring and attempt to use this substring to correct encodings in the encoding lookback buffer. One of the following three sub-steps is performed:

(a) If $v_s \leq v_m$, the matching substring overlaps only the previously unencoded suffix *V*; it cannot be used to correct encodings already in the buffer. If $v_s < v_m$, encode the substring V $[v_s, v_m)$ using an add command.

Encode the substring $V\,[v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Output the command(s) to the buffer. Set v_s to $v_m + l$.

(b) If $v_m < v_s < v_m + l$, the matching substring overlaps both the encoded prefix and the unencoded suffix of V. Perform tail correction. That is, attempt to correct encodings from the tail of the buffer by integrating commands into the new copy command. All whole and partial add commands and all whole copy commands that encode the substring $V\,[v_m, v_s)$ can be integrated into the new copy command that also encodes the substring $V\,[v_s, v_m + l)$. Delete from the buffer all commands that were wholly integrated. Output the new copy command to the buffer. Set v_s to $v_m + l$.

(c) If $v_m + l \leq v_s$, the matching substring overlaps only the existing encoded prefix of V. Perform general correction. That is, perform binary search in the buffer to find the commands that encode the substring $V\,[v_m, v_m + l)$ and correct sub-optimal encodings when possible. (In this case, v_s does not change.)

(7) Set v_c to $\max(v_m + l, v_c + 1)$ and r_c to $\max(r_m + l, r_c + 1)$; that is, we set the new values of these pointers to the offsets just following the matching substring, but we also require these pointers to advance by at least 1. Return to Step (3).

(8) All of the input has been processed. Flush all commands from the buffer to the delta encoding. If $v_s < |V|$, encode the substring $V\,[v_s, |V|)$ with an add command. Terminate the algorithm.

Figure 2.3 Pseudocode for the correcting one-pass algorithm

2.3.4 – The Correcting 1.5-Pass Algorithm

The correcting 1.5-pass algorithm can be seen as a reformulation of the greedy algorithm. The main difference is that it encodes the first matching substrings found, rather than searching exhaustively for the best matching substrings. While both algorithms make a pass over the reference string computing footprints and storing information in the hash table, the greedy algorithm stores, for each footprint, all offsets having that footprint; whereas the correcting 1.5-pass algorithm stores only the first such offset encountered. Hence it's linear running time.

The pseudocode of this algorithm follows.

Given a reference string *R* and a version string *V*, generate a delta encoding of *V* as follows:

(1) For each offset *a*, in input string *R* in the interval [0, |R| - p], generate the footprint F_R (a, a + p). For each footprint generated, if the entry of H_R indexed by that footprint is empty, store the offset in that entry:

for a = 0, 1, …, |R| - p: if H_R [F_R (a, a + p)] = empty then H_R [F_R (a, a + p)] to a.

(2) Start v_c and v_s at offset zero.

(3) If v_c+ p > |V| go to Step (8). Otherwise, generate a footprint [F_V (v_c, v_c + p)] at v_c.

(4) If H_R [F_V (v_c, v_c + p)] $\neq$ empty, check that the seed in *R* at offset H_R [F_V (v_c, v_c + p)] is identical to the seed in *V* at offset v_c. If matching seeds are found, continue at Step (5). Otherwise, increment v_c by one and repeat Step (3).

(5) Extend the matching substring forwards and backwards as far as possible from the matching seed. Set v_m and r_m to the start of the matching substring in *V* and

R, respectively, and set l to the length of this substring. Note that $v_s \leq v_c < v_m + l$ because v_c never decreases, and because the match originated at the seed at offset v_c.

(6) Encode the match and attempt to correct bad encodings. The following two sub-steps are identical to sub-steps (6a) and (6b) in the correcting one-pass algorithm (sub-step (6c) cannot occur here, because $v_s < v_m + l$):

  (a) If $v_s \leq v_m$: If $v_s < v_m$, encode the substring $V[v_s, v_m)$ using an add command. Encode the substring $V[v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in *R*. Output the command(s) to the buffer.

  Set v_s to $v_m + l$.

  (b) if $v_m < v_s$: We have noted in Step (5) that $v_s < v_m + l$. Attempt to correct encodings from the tail of the buffer. Delete from the buffer all commands that were wholly integrated. Output the new copy command to the buffer. Set v_s to $v_m + l$.

(7) Set v_c to $v_m + l$ and return to Step (3).

(8) All of the input has been processed. Flush all commands from the buffer to the delta encoding. If $v_s < |V|$, encode the substring $V[v_s, |V|)$ with an add command. Terminate the algorithm.

Figure 2.4 Pseudocode for the correcting 1.5-pass

2.4 Conclusion

This chapter reviewed recent differencing algorithms that operate at a fine granularity, make no assumptions about the format or alignment of input data and in practice, run in linear time, use constant space, and give good compression. In the next chapter, we describe a *test-bed* of differential algorithms that combines ideas from the greedy differencing algorithm and the correcting one-pass differencing algorithm and provides an environment that allows us to obtain experimental results on the compression performance of the proposed algorithm versus the algorithms described in [1].

CHAPTER III

A Test-Bed

As has been seen in the previous chapters, several differencing algorithms have been written and implemented in order to achieve good compression results while seeking the best possible time and space complexities. This chapter introduces our contribution in this regards, which is a test-bed for a constant space, linear time differencing algorithms.

The test-bed is based on the idea that if the *greedy* algorithm yields optimal compression results while the *one-pass* algorithm maintains linear time and constant space, an interesting approach would be to find an algorithm that would give better compression results then the one-pass while maintaining linear time and constant space. So the test-bed is an environment that allows a tradeoff between the greedy and the one-pass algorithms. Also, the test-bed is important because it makes it possible to test new algorithms as well as the algorithms described in [1] and gives experimental results on their compression performance, hence the term *test-bed.* Future work on differencing algorithms can be experimented using this test-bed as well. This algorithm allows users to get very good compression results without having to sacrifice time and space complexities, which is quite a burden with large data inputs; as it maintains the bound on complexities without having to accept poor compression results.

3.1 – The Algorithm

The algorithm implementation relies on two major steps. In the first step, the algorithm builds two hash tables, called H_R and H_V, for the reference file and the version file, respectively. The size of the hash tables is determined by the number of entries they have, which is the number of footprints generated from R and V, respectively. In the second step, the algorithm scans forward in both input strings R and V reading strings of constant length *p*, summarizes those seeds by footprinting them and then storing their offsets into H_R and H_V, respectively, at their generated footprint. Each footprint (in H_R or H_V) is an entry into the hash table and each footprint holds a linked list of constant size *L* where the offsets are stored. The footprints in the hash tables are used to detect matching seeds, and when this happens, we try to extend the match as far as possible in both R and V. Two different seeds might yield the same footprint hence the need to store many offsets at one given entry. Unlike the algorithms described in [1], this algorithm maintains two linked lists, one for H_R and one for H_V; having each a constant length *L*. All offsets scanned are stored and when the size of the linked list is exceeded, we overwrite the offset that has been *used*; this means that the string at that offset has been either *copied* or *added* therefore has been accounted for and its offset can be overwritten. This ensures that no offsets are overlooked. If no *used* offset is found on the linked list, the algorithm overwrites the oldest one inserted. Pseudocode for the test-bed algorithm follows.

Pseudocode for the test-bed algorithm

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) Create empty hash tables, H_R and H_V, for R and V, respectively. Initially, all entries are empty.

(2) Start pointers r_c, v_c, and v_s at offset zero. Pointer v_s marks the start of the suffix of V that has not been encoded.

(3) If $v_c + p > |V|$ and $r_c + p > |R|$ go to Step (8). Otherwise, generate footprint F_V $(v_c, v_c + p)$ when $v_c + p \leq |V|$ and footprint F_R $(r_c, r_c + p)$ when $r_c + p \leq |R|$.

(4) At each footprint value maintain a linked list of constant size *L* at H_R and H_V of all offsets that hashed to this value. When the size of the linked list is exceeded, overwrite the offsets flagged as used. If no flagged offset exists, overwrite the oldest one on the list.

(5) For footprints F_V $(v_c, v_c + p)$ and F_R $(r_c, r_c + p)$:

(a) If there is a hash table entry at the footprint value in the other string's hash table, the algorithm has found a likely matching substring. For example, $H_V [F_R (r_c, r_c + p)] \neq$ empty indicates a likely match between the seed at offset r_c in *R* and the seed at offset $H_V [F_R (r_c, r_c + p)]$ in *V*. In this case set r_m to r_c and v_m to $H_V [F_R (r_c, r_c + p)]$ to the start offsets of the potential match. Check whether the seeds at offsets r_m and v_m are identical because this hashing function is not ideal. If the seeds prove to be the same, matching substrings have been found. If this is the case, continue at Step (6) to extend the match (skipping the rest of Step (5)). Symmetrically, if $H_R [F_V (v_c, v_c + p)] \neq$ empty, set v_m to v_c and r_m to $H_R [F_V (v_c, v_c + p)]$. If the seeds at offsets r_m and v_m are identical, continue at Step (6) to extend the match. At this point, no match starting at v_c or starting at r_c has been found. Increment both r_c and v_c by one, and continue hashing at Step (3).

(6) At this step, the algorithm has found a matching seed at offsets v_m and r_m. The algorithm matches symbols forward in both strings, starting at the matching seed, to find the longest matching substring starting at v_m and r_m. Let l be the length of this substring.

(7) If $v_s <= v_m$, encode the substring $V\ [v_s, v_m)$ using an add command containing the substring $V\ [v_s, v_m)$ to be added. Encode the substring $V\ [v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Set v_s to the offset following the end of the matching substring, that is, set v_s to $v_m + l$. Set the flag of the offset of the substring encoded in both H_R and H_V to *used* and *coded* respectively. These are the nodes that will be overwritten when the linked list size is exceeded.

(8) Set r_c and v_c to the offset following the end of the match in R and V respectively {r_c is allowed to move backwards in the file to the matching offset since a matching offset in H_R may exist before the current pointer r_c.} That is set r_c to $r_m + l$. and v_c to $v_m + l$ {provided $v_s < v_m$.}

   (a) If $v_m < v_s$, increment v_c by one, and continue scanning {v_c is not allowed to move back in the file.}

   Return to hashing again at Step (3).

(9) All input has been processed. Terminate the algorithm.

Figure 3.1 Pseudocode for the Test-Bed Algorithm

The implementation of this algorithm is done using C^{++} and can be found in the Appendix.

3.2 – Time and Space Complexity

In this section we prove the following theorem.

THEOREM 5.1. *The test-bed differencing algorithm runs in time O (n) and Space O (1), where n is the total length of the input strings.*

PROOF.

The space bound is clear. At all times, the algorithm maintains two hash tables, each of which has q entries, having each a linked list of constant length L. This ensures a bound on the size of the hash tables. Except for the hash tables, the algorithm uses constant space. So, the total space used by the algorithm is $O(q * L)$ (= $O(1)$ if q and L are constants).

The time bound can be proved as follows. Initially, during Steps (1), the algorithm takes time O (q). In the subsequent steps, we follow the run of the algorithm and bound the time used in terms of the amount that the pointers r_c and v_c advance. When r_c and v_c are advancing in hashing mode, before a match is found, the algorithm uses time O (p) each time that the pointers advance by one. When a matching seed is found, let $M = \max(v_m - v_c, r_m - r_c)$. Because either $v_m = v_c$ or $r_m = r_c$, then the total time spent in hashing mode is O (pM). The number of non-empty hash table entries at this point is at most 2M, M in H_V and M in H_R. The match extension step takes time O (l). The encoding step takes time O ($v_m - v_c$); that is O (M). After a match, the pointers v_c and r_c are reset to the end of the match; let $v_c = v_m + l$ and $r_c = r_m + l$ be the values of v_c and r_c after this is done at one given period. It follows that v_c and r_c are advanced by a total net amount of at least $M + 2l$ during the period. The time spent in the period is $O(pM + l)$. Let M^* be the sum of all M_i. Because the pointers can advance by a total net amount of at most $n = |R| + |V|$ during the entire run, then $M^* + 2l \leq n$. It follows that the total time is $O(np + q) = O(n)$.

CHAPTER IV

Experimental Results

The tests included in this chapter are conducted over text files of different sizes varying between 1K and 100K. For each file, we apply the following steps:

For a given set of files of size *s*, run the test-bed as follows:

1. Let the seed's length *p* vary from 1 to 16. For each value of *p* go to Step (3)
2. Let the linked lists' sizes *L* and *L'* of H_R and H_V respectively vary from 1 to ∞. Values are chosen randomly; in our tests we chose L and L' to have this set of values {1, 5, 10, 20, 30, 40, ∞}, where ∞ takes integer numbers big enough, such as *100,000*. For each value of {L, L'} go to Step (3).
3. For a constant L and L' and a fixed length p, generate the total number of *copy* commands {number of copies} and the total length of strings encoded with an *add* command {total length added.}
4. Compute the total cost of running the algorithm on the file. Total cost = total copies + total length added.
5. The average cost per set of files of size *s* is then computed.

4.1 – Tables and Graphs

The tests results are displayed in a table showing the average cost per set of files of size *s*. To compute the cost per file, a computation of the number of copy commands, the number of add commands, and the total length of the strings added is done for each given value of p and for different values of *L*. The total cost is the summation of the total number of copies and the total length added. The average cost per file size is then computed.

4.1.1 – Files of Size 1K

- **Average Cost Tables**

Table 4.1.1.1 - Average Values. P = 1

L	1	10	∞
COPIES	333	270	273
ADDS	63	87	86
TOTAL LENGTH ADDED	91	142	138
TOTAL COST	424	412	412

Table 4.1.1.2 - Average Values. P = 2

L	1	10	∞
COPIES	73	67	63
ADDS	51	50	48
TOTAL LENGTH ADDED	247	266	249
TOTAL COST	320	332	312

Table 4.1.1.3 - Average Values. P = 3

L	1	10	∞
COPIES	34	33	33
ADDS	26	26	26
TOTAL LENGTH ADDED	309	311	311
TOTAL COST	343	344	344

Table 4.1.1.4 - Average Values. P = 8

L	1	10	∞
COPIES	7	7	7
ADDS	6	6	6
TOTAL LENGTH ADDED	368	367	367
TOTAL COST	374	373	373

Table 4.1.1.5 - Average Values. P = 16

L	1	10	∞
COPIES	4	4	4
ADDS	3	3	3
TOTAL LENGTH ADDED	264	260	260
TOTAL COST	267	264	264

- **Average Cost Graph**

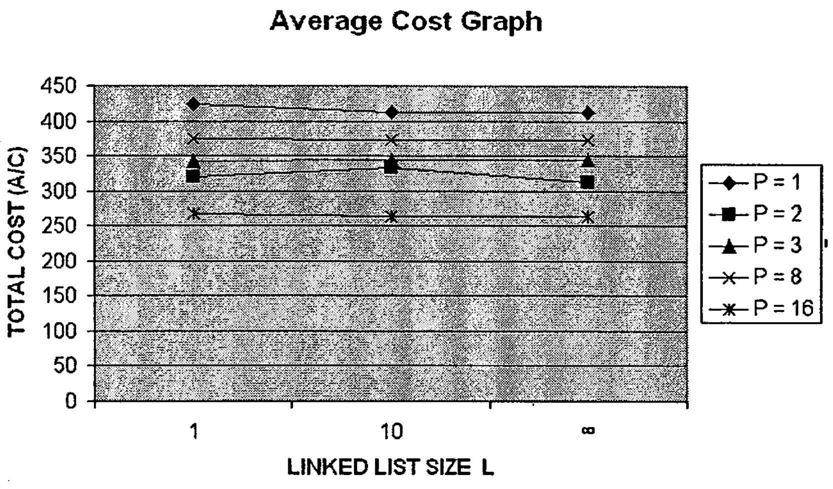

Figure 4.1 Average Cost Graph – Files of Size 1K

4.1.2 – Files of Size 8K

- **Average Cost Tables**

Table 4.1.2.1 - Average Values. P = 1

L	1	5	10	20	30	40	∞
COPIES	4783	4002	3891	3810	3756	3608	3890
ADDS	900.3	1278	1189	1225	1164	1215	1225
TOTAL LENGTH ADDED	1853	2308	2202	2309	2276	2394	2206
TOTAL COST	6635	6310	6093	6120	6031	6003	6097

Table 4.1.2.2 - Average Values. P = 2

L	1	5	10	20	30	40	∞
COPIES	1310	1186	1156	1145	1181	1194	1201
ADDS	944	866	851	855	868	872	868
TOTAL LENGTH ADDED	4544	4254	4310	4336	4246	4217	4204
TOTAL COST	5854	5440	5465	5480	5427	5412	5405

Table 4.1.2.3 - Average Values. P = 3

L	1	5	10	20	30	40	∞
COPIES	434	416	426	420	423	425	426
ADDS	346	341	355	345	348	347	348
TOTAL LENGTH ADDED	5310	5338	5304	5323	5308	5298	5294
TOTAL COST	5744	5754	5730	5743	5732	5723	5720

Table 4.1.2.4 - Average Values. P = 8

L	1	5	10	20	30	40	∞
COPIES	104	92	92	92	92	92	92
ADDS	93	86	86	87	87	87	87
TOTAL LENGTH ADDED	5947	6356	6392	6385	6385	6385	6385
TOTAL COST	6051	6448	6484	6477	6477	6477	6477

Table 4.1.2.5 - Average Values. P = 16

L	1	5	10	20	30	40	∞
COPIES	91	67	58	51	50	49	49
ADDS	44	48	46	46	46	45	45
TOTAL LENGTH ADDED	6164	6169	6232	6221	6211	6215	6215
TOTAL COST	6255	6236	6289	6272	6261	6264	6263

- **Average Cost Graph**

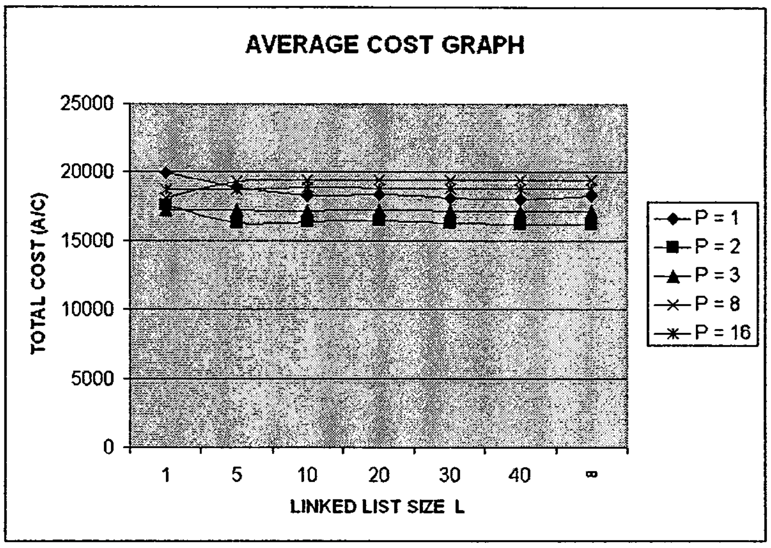

Figure 4.2 Average Cost Graph – Files of Size 8K

4.1.3 – Files of Size 30K

- **Average Cost Tables**

Table 4.1.3.1 - Average Values. P = 1

L	1	5	10	20	30	40	∞
COPIES	20416	18066	16333	15365	15076	15470	15646
ADDS	2873	2870	2999	3282	3129	3203	3146
TOTAL LENGTH ADDED	4624	4566	4734	5167	4999	5087	4911
TOTAL COST	25039	22632	21067	20532	20075	20557	20558

Table 4.1.3.2 - Average Values. P = 2

L	1	5	10	20	30	40	∞
COPIES	7336	6704	6521	6377	6277	6373	6717
ADDS	3471	3483	3526	3578	3591	3602	3477
TOTAL LENGTH ADDED	10703	11200	11514	11852	12062	11840	11079
TOTAL COST	18039	17904	18035	18229	18339	18213	17797

Table 4.1.3.3 - Average Values. P = 3

L	1	5	10	20	30	40	∞
COPIES	2806	2672	2641	2563	2514	2574	2702
ADDS	2140	2076	2069	2033	2015	2039	2082
TOTAL LENGTH ADDED	17502	17778	17838	18092	18291	18077	17608
TOTAL COST	20308	20450	20479	20655	20805	20651	20310

Table 4.1.3.4 - Average Values. P = 8

L	1	5	10	20	30	40	∞
COPIES	257	264	251	264	254	257	256
ADDS	247	254	244	257	247	249	248
TOTAL LENGTH ADDED	26744	26631	26681	26712	26665	26755	26656
TOTAL COST	27001	26895	26932	26977	26919	27012	26911

Table 4.1.3.5 - Average Values. P = 16

L	1	5	10	20	30	40	∞
COPIES	39	34	35	35	35	35	35
ADDS	30	31	32	33	33	33	33
TOTAL LENGTH ADDED	25480	26003	25975	25969	25968	25968	25968
TOTAL COST	25519	26038	26010	26004	26004	26004	26004

- **Average Cost Graph**

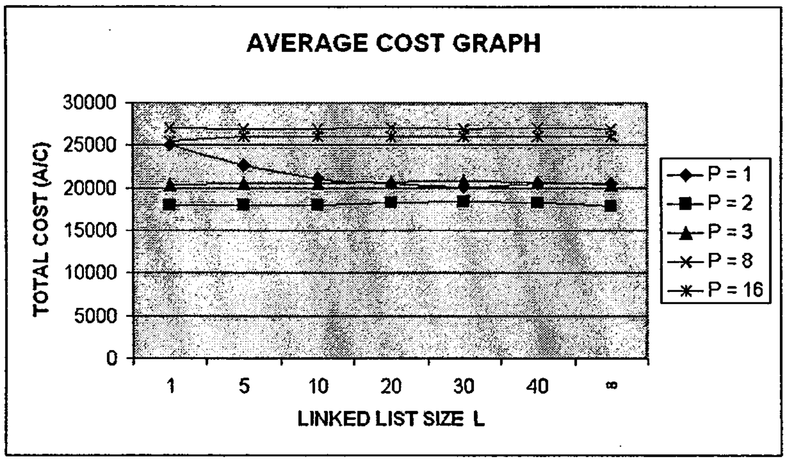

Figure 4.3 Average Cost Graph – Files of Size 30K

4.1.4 – Files of Size 40K

- **Average Cost Tables**

Table 4.1.4.1 - Average Values. P = 1

L	1	5	10	20	30	40	∞
COPIES	15932	22214	20663	19514	19077	18891	19487
ADDS	2786	4784	4780	4896	4801	4797	5018
TOTAL LENGTH ADDED	17216	8309	8294	8518	8472	8544	8688
TOTAL COST	33147	30522	28958	28032	27549	27435	28175

Table 4.1.4.2 - Average Values. P = 2

L	1	5	10	20	30	40	∞
COPIES	9691	8902	8672	8391	8324	8216	8799
ADDS	4465	4393	4431	4449	4500	4554	4376
TOTAL LENGTH ADDED	14122	14940	15137	15491	15959	16238	14824
TOTAL COST	23813	23842	23809	23882	24282	24454	23623

Table 4.1.4.3 - Average Values. P = 3

L	1	5	10	20	30	40	∞
COPIES	4154	4112	3963	3881	3828	3850	4030
ADDS	3009	2923	2891	2885	2862	2886	2903
TOTAL LENGTH ADDED	21512	21074	21500	21785	21955	21871	21240
TOTAL COST	25665	25186	25462	25666	25783	25722	25270

Table 4.1.4.4 - Average Values. P = 8

L	1	5	10	20	30	40	∞
COPIES	292	339	336	339	331	331	343
ADDS	277	326	195	328	321	319	330
TOTAL LENGTH ADDED	35242	34954	34894	34950	34976	34978	34933
TOTAL COST	35535	35292	35231	35289	35308	35309	35276

Table 4.1.4.5 - Average Values. P = 16

L	1	5	10	20	30	40	∞
COPIES	9	21	18	18	16	18	16
ADDS	8	17	17	17	15	17	15
TOTAL LENGTH ADDED	27137	35047	31530	31523	30709	31210	30709
TOTAL COST	27145	35068	31548	31542	30725	31228	30725

- **Average Cost Graph**

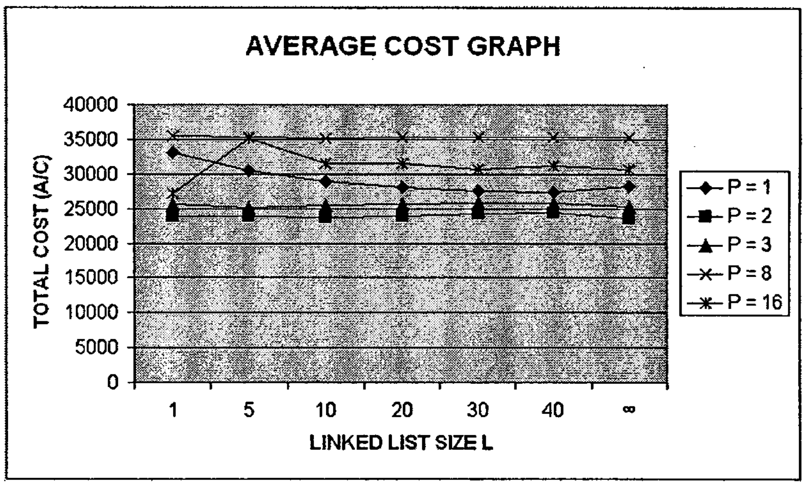

Figure 4.4 Average Cost Graph – Files of Size 40K

4.1.5 – Files of Size 70K

- **Average Cost Tables**

Table 4.1.5.1 - Average Values. P = 1

L	1	5	10	20	30	40	∞
COPIES	50198	43403	40460	39027	38439	37990	38322
ADDS	7270	8044	7934	7813	7907	7944	7994
TOTAL LENGTH ADDED	10343	11800	11760	11561	11757	11849	12199
TOTAL COST	60541	55203	52220	50588	50196	49839	50520

Table 4.1.5.2 - Average Values. P = 2

L	1	5	10	20	30	40	∞
COPIES	18695	16039	14556	16232	14922	14708	14508
ADDS	8274	8111	8031	8479	8565	8535	7948
TOTAL LENGTH ADDED	23804	27703	30911	26539	29680	30112	30949
TOTAL COST	42499	43742	45467	42770	44602	44820	45457

Table 4.1.5.3 - Average Values. P = 3

L	1	5	10	20	30	40	∞
COPIES	7561	7130	6919	6690	6579	6551	6955
ADDS	5630	5436	5356	5241	5216	5224	5352
TOTAL LENGTH ADDED	40068	40989	41414	42216	42620	42797	41261
TOTAL COST	47629	48119	48333	48906	49198	49348	48216

Table 4.1.5.4 - Average Values. P = 8

L	1	5	10	20	30	40	∞
COPIES	740	637	662	603	589	614	603
ADDS	642	606	631	584	573	598	588
TOTAL LENGTH ADDED	63763	64176	63635	64264	64413	64225	64322
TOTAL COST	64503	64813	64297	64867	65002	64839	64925

Table 4.1.5.5 - Average Values. P = 16

L	1	5	10	20	30	40	∞
COPIES	61	61	74	65	56	58	52
ADDS	53	48	50	50	46	48	43
TOTAL LENGTH ADDED	48693	52126	55478	58035	61002	57291	47301
TOTAL COST	48754	52187	55552	58100	61058	57349	47353

- **Average Cost Graph**

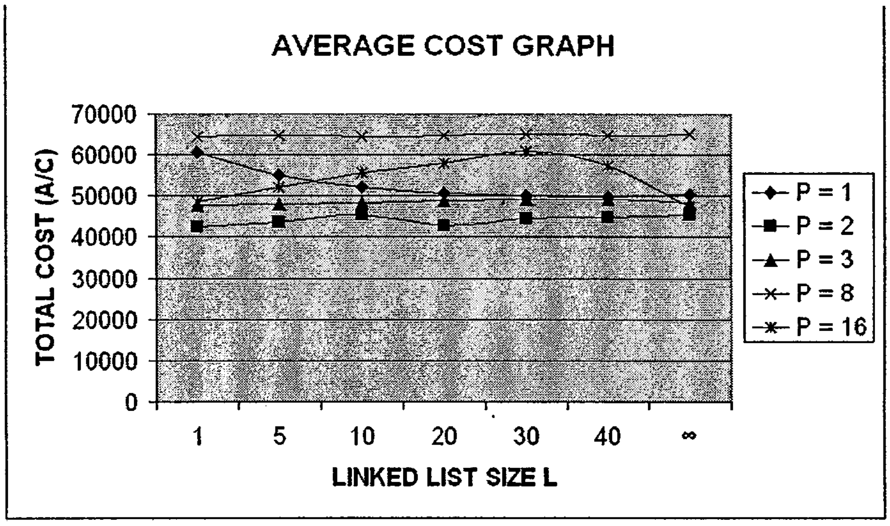

Figure 4.5 Average Cost Graph – Files of Size 70K

4.1.6 Files of Size 100K

- **Average Cost Tables**

Table 4.1.6.1 - Average Values. P = 1

L	1	5	10	20	30	40	∞
COPIES	75907	66759	62858	57717	57322	56718	57946
ADDS	10841	10986	10454	11933	10982	12124	12336
TOTAL LENGTH ADDED	16579	17989	16945	19606	17898	20443	20692
TOTAL COST	92486	84748	79803	77323	75220	77161	78639

Table 4.1.6.2 - Average Values. P = 2

L	1	5	10	20	30	40	∞
COPIES	29440	27245	26553	26197	25797	25714	26555
ADDS	11769	12239	12473	12243	12400	12412	12334
TOTAL LENGTH ADDED	35319	37491	38702	38379	39487	39326	38024
TOTAL COST	64760	64736	65255	64576	65284	65039	64578

Table 4.1.6.3 - Average Values. P = 3

L	1	5	10	20	30	40	∞
COPIES	12227	11767	11449	11306	11205	11267	11601
ADDS	8552	8341	8236	8205	8250	8293	8296
TOTAL LENGTH ADDED	58004	60669	61564	62024	62404	62069	61016
TOTAL COST	70232	72436	73013	73331	73608	73336	72618

Table 4.1.6.4 - Average Values. P = 8

L	1	5	10	20	30	40	∞
COPIES	919	1423	1363	1353	1385	1326	4558
ADDS	830	1349	1300	1294	1321	1268	3555
TOTAL LENGTH ADDED	95638	97533	98596	98638	98456	98735	87155
TOTAL COST	96556	98956	99958	99991	99841	100061	91713

Table 4.1.6.5 - Average Values. P = 16

L	1	5	10	20	30	40	∞
COPIES	43	109	69	87	93	76	77
ADDS	41	75	67	83	89	72	73
TOTAL LENGTH ADDED	95977	96347	91292	95227	95186	88528	88524
TOTAL COST	96020	96456	91362	95314	95278	88604	88600

- **Average Cost Graph**

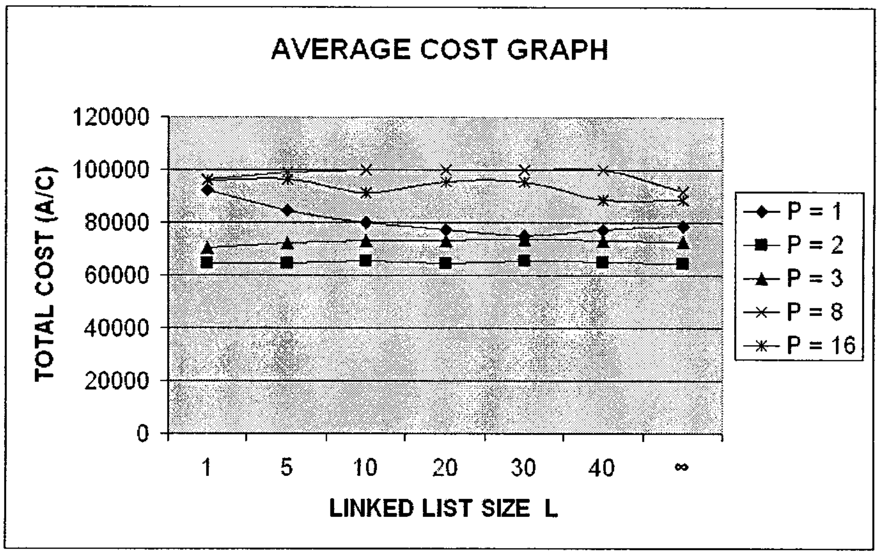

Figure 4.6 Average Cost Graph – Files of Size 100K

4.2 Recommendations

Based on the above set of experiments, some recommendations can be made as to the values of *p* and *L* that relatively yield good compression results. Next, and for each set of files of size *s*, we highlight which values of *p* and *L* give good compression results while maintaining linear time and constant space.

4.2.1 – Files of Size 1K

The experimental results show that a good average compression cost of files of size 1K, is achieved for a seed's length *p = 2* and for a constant linked list size L = 40. The values are as follows:

Good Average Cost: *264* for *p = 16* and *L = 10*

4.4.2 – Files of Size 8K

A good average compression cost is achieved for the following values:

Good Average Cost: *5412* for *p = 2* and *L = 40*

4.2.3 – Files of Size 30K

A good average compression cost is achieved for the following values:

Good Average Cost: *17904* for p = *2* and L = *5*

4.4.4 – Files of Size 40K

A good average compression cost is achieved for the following values:

Good Average Cost: *23809* for P = *2* and L = *10*

4.4.5 – Files of Size 70K

A good average compression cost is achieved for the following values:

Good Average Cost: *42499* for P = *2* and L = *1*

4.4.6 – Files of Size 100K

A good average compression cost is achieved for the following values:

Good Average Cost: *64576* for p = *2* and L = *20*

CHAPTER VI

Conclusion

In this thesis, we have created a test-bed for differencing algorithms that run in linear time and use constant space. We have discussed the delta encoding differencing technique. Our experimental results provided the best values for the length of the linked list and the length of the seed to be used in the hash tables as function of the input size.

References

[1] M. Ajtai, R. Burns, R. Fagin, D.D.E. Long, and L. Stockmeyer, Compactly encoding unstructured· inputs with differential compression, *Journal of the ACM* 49:3, 2002, 318-367.

[2] S. Baase and A. Van Gelder, *Computer Algorithms,* 3rd Edition. Addison Wesley, 2000.

[3] G. Banga, F. Douglis, and M. Rabinovitch, Optimistic deltas for WWW latency reduction. In *Proceedings of the 1997 USENIX Annual Technical Conference.* USENIX Association, Berkeley, California, 1997, 289-303.

[4] R. C. Burns and D.D.E. Long, In-place reconstruction of delta compressed files. In *Proceedings of the 17thAnnual ACM Symposium on Principles of Distributed Computing.* ACM, New York, 1998.

[5] D. Gusfield, *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.

[6] J. P. MacDonald, File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California, 2000.

[7] W. Miller and E. W. Myers, A file comparison program. *Softw. Pract. Exper.* 15:11, 1985, 1025-1040.

[8] J. C. Mogul, F. Douglis, A. Feldman, and B. Krishnamurthy, Potential benefits of delta encoding and delta compression for HTTP. In *Proceedings of ACM SIGCOMM'97,* ACM, New York, 1997.

[9] C. Reichenberger, Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management.* ACM, New York, 1991, 144-152.

[10] W. F. Tichy, The string to string correction problem with block move, *ACM Trans. Comput.* 2:4, 1984, 309-321.

APPENDIX

A TEST-BED FOR LINEAR TIME CONSTANT SPACE DIFFERENCING ALGORITHMS

```
#include "list.cpp"
#include "Node.h"
#include "Vlist.cpp"
#include "VNode.h"
#include "declarations.h"
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include <limits.h>


/*
                                        Main()
*/

main()
{


        ifstream readReference("reference.txt", ios::in), readVersion("version.txt",
ios::in);



        if(!readReference)
        {
            cerr<<"File could not be opened"<<endl;
            exit(1);
        }//endif

        if(!readVersion)
        {
            cerr<<"File could not be opened"<<endl;
            exit(1);
        }//endif



        readReference.read((char*)&current, p);

        readVersion.read((char*)&versionstring, p);

        //BUILD THE HASH TABLES INCREMENTALLY
        proceed(readReference, readVersion, HR,HV,current, versionstring,rc, vc, p);


        cout<<"\n\n\n\t n----------------- Total # of Copies for this File are:
        '"<<COPYCOUNTER<<"'"<<endl;

        cout<<"\n\n\n\t n----------------- Total # of Adds for this file are:
        '"<<ADDCOUNTER<<"'"<<endl;

        cout<<"\n\n\n\t n----------------- Total Length of the Strings Added for this file
are: '"<<TOTALLEN<<"'"<<endl;

        cout<<"\n\n\n FILES OVER!!!!!   PROCEED() IS OVER";
        return 0;
}//endofmain()

/*
                                    End of Main()
*/


/*
                                    Proceed()
*/



void proceed(ifstream readReference, ifstream readVersion, List HR[LISTSIZE], VList
HV[LISTSIZE], Seed current, VerSeed versionstring, int rc, int vc, int p)
```

```
{

    while ((!readReference.eof())
            && (!readVersion.eof()))

    {


            //cout<<"\n\n ------------------------------------------------------------\n";
            //getchar();

    //      cout<<"\n Current REFERENCE String is: ";


            outputline(cout, current);


            //rc is the starting offset of each scanned string in the file
            rc = readReference.tellg() - p;
            cout<<" At Offset RC = "<<rc;
            cout<<"  ";

            //Compute footprint of the reference seed
            unsigned long  fprintR = footprint(current,rc, p);
            cout<<" With Footprint =  "<<fprintR;

            HR[fprintR].insertAtFront(rc);          //insert rc into hash table HR
                                                    //at its footprint fprintR

            cout<<"\n\n Current VERSION String is:  ";
            outputline2(cout, versionstring);

            vc = readVersion.tellg() - p;  //Vc is the starting offset of a scanned
                                           //string in the file
            cout<<" At Offset VC = "<<vc;

            //compute a Version seed's footprint
            unsigned long  fprintV = footprint2(versionstring,vc, p);
            cout<<" With Footprint = "<<fprintV<<endl;
            HV[fprintV].VinsertAtFront(vc);


            //Check for each seed scanned in the version file whether its footprint
            //exists in the hash table of R.
            //And simultaneously, for each seed scanned in the reference file,
            //check whether its footprint exists in the hash table of V.
```

```
//This ensures we are maintaining a window into the past for finding
//transposed seeds

//   1)  Search for fooptrint fprintV in hash table HR[]

//if firstptr != 0 then fprintV exists in H_R
if (!HR[fprintV].isEmpty())
{
        cout<<" \n\n Footprint from HV[] '"<<fprintV<<"' was found in
        HR[]. \n\n\t Examine offsets in HR[]: \n"<<endl;

                Rptr = HR[fprintV].Fptr(); //Rptr = first node on the list
                                                  //of HR[fprintV]
                int offR = HR[fprintV].getOffset(Rptr);//offR = offset
                                                       //stored at Rptr

        //now go in V to the offset stored at HV[fprintV]
        Vptr = HV[fprintV].VFptr();        //VNode = first node on the
                                           //list of HV[fprintV]
        int offV = HV[fprintV].VgetOffset(Vptr);        //offV = offset
                                           // stored at Vptr


        while (Rptr != 0)     //while an offset exists on the next node of
                              //the list at HR[fprintV]
        {
                //search the list for a node that has not yet been encoded
                //i.e. used=false.
                //if no such node is found encode the used ones on the
        //list.


                offR = HR[fprintV].getOffset(Rptr);        //get next offset
                                                           //in the list

                rm = offR;    //a possible match exists: set rm = offR

                vm = offV;    //a possible match exists; set vm = offV;

                if (vs <= vm )
                {
                        readReference.seekg(offR); //go to offR in R
                        readVersion.seekg(offV);   //go to offV in V
```

```
            //scan from both files a string of length p=3
              readReference.read((char*)&indexedseed, p);
              readVersion.read((char*)&indexedverseed,p);


              Encode(Rptr,Vptr,readReference, readVersion, HR,
              HV, indexedseed, indexedverseed,rc,vc, rm, vm, p,
              fprintR, fprintV);


        //keeping record of the longest match only.
        if ((len >= leng) && (len != 0))

        {
              for (int cp = 0; cp < len; cp++)
                    tempstring[cp] = REFlongestmatch[cp];

              tempstring[len] = '\0';


              matchingR = rm;
              matchingV = vm;
              longestRptr = Rptr;
              longestVptr = Vptr;

              leng = strlen(tempstring);
        }//endif()
        else ;                //if length of a new string is smaller
                              //than its predecessor disregard it
        }//endif(vs<vm)
        else;                 //if vm<vs don't accept it. We only
                              //move forward in V

        Rptr = HR[fprintV].nPtr(Rptr);     //next pointer
        offR = HR[fprintV].getOffset(Rptr);       //next offset
}//endwhile()

if ((leng > 0) && (vm >= vs))
{
        //Before Copy-encoding a matching seed, Add() the
        //string [Vs,Vm)

        int l = vm-vs;  //l = length of string [Vs, Vm-1]

        if (l == 0)
              cout<<"\n\n\t No String TO ADD.";
```

```
                else
                {
                        int curr = readVersion.tellg();
                        int Rcurr = readReference.tellg();

                        readVersion.seekg(vs);      //position pointer at Vs
                        int initial = readVersion.tellg(); //vs value

                        readVersion.read((char*)&vstring,l);      //scan
                                                                  //seed [Vs,Vm)

                        ADD(vstring,l,vs);
                        int stln = strlen (vstring.VerSubstring);

                        for (int i = 0; i <= stln; i++)
                                vstring.VerSubstring[i] = '\0';

                        readVersion.seekg(curr);    //reposition the pointer
                }//endof Else()

                COPY(tempstring, leng,matchingR);       //matchingR =
                                                        //offset of the longest
                                                   //string stored in REFlong

                //Set Flags in H_R & H_V of the encoded offsets
                HR[fprintR].setflag(longestRptr,matchingR);

                HV[fprintV].Vsetflag(longestVptr,matchingV);

                vs = matchingV + leng;// + 1;
                vc = vs + p - 1;

                rc = matchingR + leng + p-1;        //rc moves backwards

                readVersion.seekg(vc);
                readReference.seekg(rc);
        }//endif(leng>0)
        else ;

}//endif(!HR[fprintV].isEmpty())
else
{
        cout<<"\n\n FOOTPRINT from HV[] NOT FOUND IN
        HR[]"<<endl;
}
```

```
        for (int e = 0; e <= SIZE; e++)        //re-initialize tempstring[]
                tempstring[e] = '\0';

        leng = 0;                              //re-initialize leng

        for (int ee = 0; ee <= SIZE; ee++)  //re-initialize REFlongestmatch
                REFlongestmatch[ee] = '\0';

        len = 0;


// 2)   Search for fooptrint fprintR in hash table HV[]

        //if firstptr != 0 then fprintR exists in Hv
        if (!HV[fprintR].VisEmpty())
        {
                cout<<" \n\n Footprint from HR[] '"<<fprintR<<"' was found in
                HV[]. \n\n\t Examine offsets in HV[]: \n"<<endl;

                //get first offset from Hv at fprintR in the first node of the list;
                VVptr = HV[fprintR].VFptr();
                int ofV = HV[fprintR].VgetOffset(VVptr);

                RRptr = HR[fprintR].Fptr();
                int ofR = HR[fprintR].getOffset(RRptr);

                while (VVptr != 0)
                {
                        //search the list for a node that has not yet been encoded
                        //i.e. used=false.
                        //if no such node is found we encode the used ones on the
                        //list.
                        vm = ofV;               //a possible match exists
                        rm = ofR;


                        //Go in R to the offset stored at HR[fprintR]
                        if (vs <= vm)
                        {
                                readVersion.seekg(ofV);
                                readReference.seekg(ofR);

                                // scan from R &V a string of length p=3
                                readReference.read((char*)&Rseed, p);
                                readVersion.read((char*)&Vseed,p);
                Encode(RRptr,VVptr,readReference, readVersion,
                HR, HV, Rseed, Vseed,rc,vc, rm, vm, p, fprintR,
                fprintV);

            if (len >= ln)
            {
                for (int ss = 0; ss < len; ss++)
                    returned[ss] = REFlongestmatch[ss];

                returned[len] = '\0';

                matchingR = rm;
                matchingV = vm;
                longestRptr = RRptr;
                longestVptr = VVptr;

                ln = strlen(returned);//update length of returned[]
            }
            else ;
        }//endif(vs<=vm)
        else;

        //VVptr =next node-offset in the list at fprintR
        VVptr = HV[fprintR].VnPtr(VVptr);
        ofV = HV[fprintR].VgetOffset(VVptr);
}//endwhile()

if ((leng > 0) && (vm >= vs))
{
        //Before Copy-encoding a matching seed, Add() the
        //string [Vs,Vm)
        int l = vm-vs;          //l = length of string [Vs Vm-1]

        if (l == 0)
                cout<<"\n\n\t No String TO ADD.";
        else
        {
        int curr = readVersion.tellg();
        int Rcurr = readReference.tellg();

        readVersion.seekg(vs);       //position the file pointer at VS

        int initial = readVersion.tellg();

        readVersion.read((char*)&vstring,l);      //get string [Vs,Vm)
```

```
            ADD(vstring,l,vs);
            int stlng = strlen(vstring.VerSubstring);

            for (int i = 0; i <= stlng; i++)
                    vstring.VerSubstring[i] = '\0';

            readVersion.seekg(curr);            //reposition the pointer
            }//endof Else()

            COPY(returned, ln,matchingR);

            //Set flags in both H_R & H_V
            HR[fprintR].setflag(longestRptr,matchingR);
            HV[fprintV].Vsetflag(longestVptr,matchingV);

            vs = matchingV + ln;//ln = length of the matching seed
            vc = vs + p - 1;

            rc = matchingR + ln + p -1;         //rc moves backwards

            readVersion.seekg(vc);              //re-position pointer inV
            readReference.seekg(rc);    //re-position pointer in R

        }//endif(leng>0)
        else
                        ;
}//endif(!HV[fprintR].isEmpty())
else
{
        cout<<"\n\n FOOTPRINT from HR[] NOT FOUND IN HV[]"<<endl;
}

for (int f = 0; f <= SIZE; f++)             //re-initialize
        returned[f] = '\0';

ln = 0;                                     //re-initialize

for (ee = 0; ee <= SIZE; ee++)                      //re-initialize
REFlongestmatch[ee] = '\0';

len = 0;


        int vv = readVersion.tellg() - p + 1;
```

```
                readVersion.seekg(vv);

                int rr = readReference.tellg() - p + 1;
                readReference.seekg(rr);

                readReference.read((char*)&current, p);
                readVersion.read((char*)&versionstring, p);

        }//endofwhile()----------> end of file is reached.
}//endof Proceed()




/*
                                        Encode()
*/



void  Encode(Node * rrptr, VNode * Vvptr,ifstream readReference, ifstream
readVersion, List HR[LISTSIZE], VList HV[LISTSIZE], Seed current, VerSeed
versionstring, int rc, int vc, int rcm, int vcm, int p, int fprintR, int fprintV)
{

        ExtendMatch stretch;
        ExtendMatch2 grow;

        //Check if two strings match

        cmp = CompareSeeds(current, versionstring);

        if(vcm >= vs)
        {
                if (cmp == 0)                   //The two strings match
                {
                        rcmatch = rcm;
                        vcmatch = vcm;

                        cout<<"\n\n\t            MATCHING SEEDS FOUND !!!";
```

```
for (int m=0; m <3;m++)
{
        matchREF[m] = current.substring[m];
        matchVER[m] = versionstring.VerSubstring[m];
}//endfor


strcpy(REFlongestmatch,matchREF);
strcpy(VERlongestmatch,matchVER);

int off;
int Voff;

off = readReference.tellg() - p; //offset of the matching seed in R
Voff = readVersion.tellg() - p; //offset of the matching seed in V


int Vextensionpoint = vcm + p;
int Rextensionpoint = rcm + p;
readReference.seekg(Rextensionpoint);
readVersion.seekg(Vextensionpoint);


while (1)
{
        readReference.read((char*)&stretch, 1);  //check next
                                                   //character
        readVersion.read((char*)&grow, 1); //check next
                                            //character

        //if next characters don't match, extension not possible.
        if (stretch.Rlongest[0] != grow.Vlongest[0])
        {
                break;
        }
        else
        {
                strncat(REFlongestmatch,stretch.Rlongest,1);
                strncat(VERlongestmatch,grow.Vlongest,1);


                if((readReference.eof()) || (readVersion.eof()))
                {
                        break;
                        }//endif()
                    }//endelse()
                }//endWhile()
                //len =  length of the longest matching string
                len = strlen(REFlongestmatch);


            }//endif(cmp==0)
            else                            //strings don't match
            {
                cout<<"\n\n\n\n\t  STRINGS DON'T MATCH !!! ";

                rc = rc + p;                //advance rc by p since we scan strings from
                                            //current location - length + 1
                readReference.seekg(rc);

                vc = vc + p;   //advance vc by p
                readVersion.seekg(vc);
            }//endElse()
        }//endif(vcm>=vs)
            else ;          //vm < vs, disregard.
}//endof Encode()



/*
                                footprint()
*/



unsigned long footprint(Seed curr, int r, int p)
{
        F=0;   //the footprint of a scanned seed

        int q = 1001;  // number of entries in the hash table


        //r = current offset & corresponds to j=0 of the seed stored in substring[j] of
        //length p=3
        //and i <= r + p - 1 corresponds j<= p-1.

        int j = 0;
        while (j <= p - 1)
                if(j<=p-1)
                {
                        int i = r;
                        while (i <= r + p - 1)
                        {
                                F += pow(127,r + p - 1 - i) * curr.substring[j];

                                i++;
                                j++;
                        }//endwhile()

                        j++; //j>2 : Get next string
        }//endFor()

        return F % q;
}//end of footprint()




/*
                                                Footprint2()
*/




unsigned long footprint2(VerSeed S, int r, int p)
{
        Fv=0;   //the footprint of a scanned seed

        int q = 1001;  // number of entries in the hash table


        //r=current offset & corresponds to j=0 of the seed stored in substring[j] of
        //length p=3
        //and i <= r + p - 1 corresponds j<= p-1.

        int j = 0;
        while (j <= p - 1)
                if(j<=p-1)
                {
                        int i = r;
```

```
                    while (i <= r + p - 1)
                    {
                            Fv += pow(127,r + p - 1 - i) * S.VerSubstring[j];


                            i++;
                            j++;
                    }//endwhile()

                    j++; //j>2 : get the next string
        }//endFor()

        return Fv % q;
}//end of footprint2()




/*
                                    Outputline()
*/




void outputline(ostream &out, Seed s)
{
        out<<"'"<<s.substring<<"'";

}//endof outputline()




/*
                                    Outputline2()
*/



void outputline2(ostream &print, VerSeed vs)
{
        print<<"'"<<vs.VerSubstring<<"'";

}//endof outputline2()




/*
                                        CompareSeeds()
*/




int CompareSeeds(Seed cs, VerSeed Vcs)
{

        int comp;
        comp = strcmp(cs.substring, Vcs.VerSubstring);
        return comp;
}//endofCompareSeeds()

/*
                                             Copy()
*/


void COPY (char * REFstr, int leng, int addr)
{
        COPYCOUNTER++;

        cout<<"\n\n\n------COPY MATCHING REFERENCE STRING------";

        cout<<"\n\n COPY  '"<<REFstr<<"'";
        cout<<"\n (C,"<<leng<<","<<addr<<")"<<endl;

        cout<<"\n TOTAL LENGTH ENCODED WITH A COPY() is: "<<leng<<endl;
        cout<<"\n TOTAL NUMBER OF COPIES = "<<COPYCOUNTER<<endl;
}//end of Copy()
```

```
/*
                                    Add()
*/

void ADD(VerSeed Vstr,int LEN,int q)
{
        ADDCOUNTER++;
        TOTALLEN += LEN;
        cout<<"\n\n------ADD NON-MATCHING VERSION STRING------";

        cout<<"\n\n (A,"<<LEN<<","<<Vstr.VerSubstring<<")"<<endl;

        cout<<"\n TOTAL LENGTH ENCODED WITH AN ADD() is: "<<LEN<<endl;
        cout<<"\n TOTAL NUMBER OF ADDS = "<<ADDCOUNTER<<endl;
}//end of add()
```

```
#ifndef NODE_H
#define NODE_H

class Node{
        friend class List;
public:
        Node(const int &);
        int getdata();
        int NodeIsFlagged(Node *);          //checks if a given node has been flagged
        bool used;

private:
        Node * nextPtr;
        int data;
};

Node::Node(const int & info)
   :data(info),nextPtr(0), used(false) { }

int Node::getdata()
{
        return data;
}



#endif
```

```
//LIST.H
//This is the header file for the list class.

#ifndef List_H
#define List_H
#include"node.h"

class List{
        friend class Node;
public:
        List();
        void insertAtFront(const int);
        bool isEmpty();
        void print();
        Node * getNewNode(const int &);

        bool setflag(Node *,int);     //setflag() takes an encoded offset as
                                      //argument & sets its node to used=true.
        int getOffset(Node *);


        Node * Fptr();
        Node *nPtr(Node*);

private:
        Node * firstPtr;
        Node * lastPtr;
        int count;
};
#endif
```

```
//LIST.CPP

#include<iostream.h>
#include<cassert>
#include "node.h"
#include "list.h"
#include <stdio.h>
#include <iomanip.h>

const int MAXCOUNT = 5;

List HR[2311];

List::List()
  :firstPtr(0),lastPtr(0), count(0){ }

int List::getOffset(Node * ptr)

{
        if(ptr != 0){
                int a = ptr->data;
                return a;
        }
        else
                return -1;
}


Node * List::nPtr(Node *ptr)
{

      return ptr->nextPtr;

}


Node * List::Fptr()
{
        return firstPtr;
}


bool List::setflag(Node * rptr, int offset)
{
        while(rptr->data != offset)
                rptr=rptr->nextPtr;
        rptr->used = true;
        int nodeoffset = rptr->getdata();
        cout<<"\n\n\t FLAG OF NODEOFFSET '"<<nodeoffset<<"' IS SET TO
        TRUE"<<endl;
        return true;                    //node is flagged as used.

}//endofsetflag()


void List::insertAtFront(const int value)
{

        Node  * newPtr = getNewNode(value);
        Node * p = firstPtr;
        if(isEmpty())
        {
                firstPtr = lastPtr = newPtr; count = 1;
        }
        else
        {
                if (count < MAXCOUNT)
                {       //while there is still empty nodes in the list
                        Node * f = firstPtr;

                        if (f != 0)
                                while (f != 0)          //Insert the offset ONLY if it's not
                                                //previously inserted i.e. no duplication
                                {
                                        if (f->data == value)
                                        {       //offset exists in list of HR, don't duplicate
                                                break;
                                        }

                                        f = f->nextPtr;
                                }
                                if (f==0)
                                {
                                        newPtr->nextPtr = firstPtr;
                                        firstPtr = newPtr;
                                        count++;
                                }//endif()
                }//endif()
```

```
else
{
        bool found;
         found = false;
         do              //(p->nextPtr != firstPtr)
         {
                 if (p->used == true)
                  {
                          cout<<"\n one used offset found in H_R ";


                          p->data = value;
                          p->used = false;
                          found = true;
         }//endif()
         else
                  p = p->nextPtr;
 }//endwhile()
 while ((p != 0) && !found);

 if (!found)             // none of the offsets has been used
 {
         if (firstPtr  == lastPtr)
         {
                  firstPtr->data = value;
         }
         else
         {
                 Node * current = lastPtr;
                 Node * previous;

                 while (firstPtr->nextPtr != current)
                 {
                          previous = firstPtr;

                         while (previous->nextPtr != current)
                                  previous = previous->nextPtr;

                          current->data = previous->data;
                          current = previous;
                 }//endwhile()

                 current->data = firstPtr->data;
                 firstPtr->data = value;
          }//endelse
          }//endif(!found)
```

```
                }//endElse()
                }//endif()
}//endInsertAtFront()


Node * List::getNewNode(const int & value)
{
        Node * ptr = new Node (value);
        assert (ptr != 0);
        return ptr;
}

bool List::isEmpty()
{
        return (firstPtr == 0);
}
void List::print()
{
        Node * Ptr = firstPtr;
        while(Ptr != 0)
        {
                cout<<Ptr->data<<" ";
                Ptr = Ptr->nextPtr;
        }//endwhile()
}//endprint()
```

```
//VNODE.H

#ifndef VNODE_H
#define VNODE_H

class VNode{
        friend class VList;
public:
        VNode(const int &);
        int getVdata();
        int VNodeIsFlagged(VNode *);            //checks if a given node is empty
        bool coded;

private:
        VNode * VnextPtr;
        int Vdata;

};

VNode::VNode(const int & Vinfo)
   :Vdata(Vinfo),VnextPtr(0), coded(false){}

int VNode::getVdata()
{
        return Vdata;
}



#endif
```

```
//VLIST.H
//This is the header file for the VList class.

#ifndef VList_H
#define VList_H
#include"Vnode.h"

class VList{
	friend class VNode;
public:
	VList();
	void VinsertAtFront(const int);
	bool VisEmpty();
	void Vprint();

	VNode * VgetNewNode(const int &);

	bool Vsetflag(VNode *,int);		//Vsetflag() takes an encoded offset as
						//argument & sets its node to used=true.
	int SearchHashTable_V(VNode *);		//search HV[] for a given footprint
	int VgetOffset(VNode *);
	VNode * VFptr();
	VNode *VnPtr(VNode*);

private:
	VNode * VfirstPtr;
	VNode * VlastPtr;
	int Vcount;
};
#endif
```

```
//VLIST.CPP

#include<iostream.h>
#include<cassert>
#include "Vnode.h"
#include "VList.h"
#include <stdio.h>
#include <iomanip.h>


const int MAXSIZE = 20;

VList HV[2311];

VList::VList()
 :VfirstPtr(0),VlastPtr(0), Vcount(0) { }

int VList::VgetOffset(VNode * Vptr)
{
        if(Vptr != 0){
                int a = Vptr->Vdata;
                return a;
        }
        else
                return -1;
}
VNode * VList::VnPtr(VNode *Vptr)
{
        if(Vptr)
      return Vptr->VnextPtr;
        else
                return 0;
}
VNode * VList::VFptr()
{
        return VfirstPtr;

bool VList::Vsetflag(VNode * vvptr,int of)
{
        while (vvptr->Vdata != of)
                vvptr = vvptr->VnextPtr;

        vvptr->coded = true;
        int VNODEOFFSET = vvptr->getVdata();
        cout<<"\n\n\t FLAG OF VNODEOFFSET '"<<VNODEOFFSET<<"' IS SET
        TO TRUE"<<endl;
        return true;            //node is flagged as coded
}//end of Vsetflag()





void VList::VinsertAtFront(const int Vvalue)
{

        VNode  * VnewPtr = VgetNewNode(Vvalue);
        VNode * p = VfirstPtr;
        if(VisEmpty())
        {

                VfirstPtr = VlastPtr = VnewPtr;
                Vcount = 1;
        }
        else
        {
                if (Vcount < MAXSIZE)
                {
                        VNode * v = VfirstPtr;

                        if (v != 0)
                                while (v != 0)
                                {
                                        if (v->Vdata == Vvalue)
                                        {//Offset exists in Hvdon't duplicate
                                                break;
                                        }
                                        v = v->VnextPtr;
                                }//endWhile()
                        if (v == 0)
                        {
                                VnewPtr->VnextPtr = VfirstPtr;
                                VfirstPtr = VnewPtr;
                                Vcount++;
                        }//endif()
                }//endifVcount<MAXSIZE)


                else
                {
                        VNode * p = VfirstPtr;
```

```
            bool found = false;
        do
        {
                if (p->coded == true)
                {
                        cout<<"\n ONE USED OFFSET FOUND ON HV[]";


                                p->Vdata = Vvalue;
                                p->coded = false;
                                found = true;
                }//endif()
                else
                        p = p->VnextPtr;

        }
        while ((p != 0) && !found);

        if (!found)                     // none of the offsets has been used
        {
                if (VfirstPtr  == VlastPtr)

                {
                        VfirstPtr->Vdata = Vvalue;
                }
                else {
                            VNode * current = VlastPtr;
                            VNode * previous;

                while (VfirstPtr->VnextPtr != current)
                {
                        previous = VfirstPtr;

                        while (previous->VnextPtr != current)
                                previous = previous->VnextPtr;

                        current->Vdata = previous->Vdata;
                        current = previous;
                }//endwhile()

                current->Vdata = VfirstPtr->Vdata;
                VfirstPtr->Vdata = Vvalue;
         }//endElse()
         }  //endif(!found)
  }//endElse()
}//endif()
```

```
}//endVInsertAtFront()


VNode * VList::VgetNewNode(const int & Vvalue)
{
	VNode * Vptr = new VNode (Vvalue);
	assert (Vptr != 0);
	return Vptr;
}



bool VList::VisEmpty()
{
	return VfirstPtr == 0;
}




void VList::Vprint()
{
	VNode * VPtr = VfirstPtr;
	while(VPtr != 0)
	{
		cout<<VPtr->Vdata<<" ";
		VPtr = VPtr->VnextPtr;
	}//endWhile()
}//endVprint()
```

```
//DECLARATIONS.H
//This file includes all the declarations of the variables, functions and structures
used //in the code.

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include <limits.h>
int p = 3;                    //length of the seed.
int TOTALLEN = 0;         //total length of the adds encoded
const int LISTSIZE = 2311;
const int SIZE = 100000;
struct Seed {
        char  substring[2000];
};
struct VerSeed{
        char  VerSubstring[2000];
};
struct ExtendMatch {
        char Rlongest[2];
};
struct ExtendMatch2 {
        char Vlongest[2];
};

/*
                              THE FUNCTIONS

*/

void proceed(ifstream, ifstream, List[LISTSIZE], VList[LISTSIZE], Seed,
VerSeed,int, int, int);

//The following functions encode a string as an Add() or Copy() command.

void  Encode(Node *,VNode *, ifstream, ifstream, List[LISTSIZE],
VList[LISTSIZE], Seed, VerSeed,int, int,int, int, int, int, int);

void outputline(ostream&, Seed);          //outputs a reference string
void outputline2(ostream&, VerSeed);    //outputs a version string
```

```
unsigned long  footprint(Seed, int, int);           //footprint() function for R
unsigned long  footprint2(VerSeed, int, int);       //footprint() function for V
int CompareSeeds(Seed, VerSeed);            // compare 2 seeds both files


void COPY (char *, int, int);               //copy a string
void ADD(VerSeed,int,int);                  //add a string



/*
                        THE VARIABLES

*/

/*
                        Variables of type Struct()
*/

Seed current, indexedseed, Rseed;
VerSeed versionstring, vstring, indexedverseed, Vseed;

ExtendMatch stretch;
ExtendMatch2 grow;


/*
                        POINTERS
 */

Node * Rptr;
Node * RRptr;
Node * longestRptr;
Node * bestRptr;
Node * Tptr;
Node * Sptr;

VNode * Vptr;
VNode * VVptr;
VNode * longestVptr;
VNode * bestVptr;
VNode * Uptr;
VNode * Zptr;


int vs = 0,rs = 0;                    //initial offset = 0 in the Version file
```

```
unsigned long  F;                    //footprint of a seed in R
unsigned long  Fv;                   //footprint of a seed in V

//int Fr=0;                                   //footprint stored in HR[]
//int FFv=0;                                  //footprint stored in HV[]

//int Rsearchresult = 0;               //the value returned by function
SearchHashTable_R(): either an 1 or -1
//int Vsearchresult = 0;               //the value returned by function
SearchHashTable_V(): either an 1 or -1


//the following arrays will temporarily hold each string currently being
manipulated
char  matchREF[3];
char  matchVER[3];

char  matchREF2[3];
char  matchVER2[3];



//the following arrays will hold the longest matching substrings to be copied
char  REFlongestmatch[SIZE];
char VERlongestmatch[SIZE];

char REFlongestmatch2[SIZE];
char VERlongestmatch2[SIZE];

char REFlong[SIZE];
char VERlong[SIZE];
char REFlong2[SIZE];

char tempstring[SIZE];
char returned[SIZE];


char STRING_TO_ADD[3];                 //temporary storage for the strings to
be Encoded with an ADD()
char ADDED_STRINGS[1000]; //           PERMANENT STORAGE of ALL
the Strings Encoded with an ADD()


int cmp;
```

```
long rc=0,rcc=0,rcmatch=0; // offset of the first character of the substring in R
long rm = 0;  //the starting offset of a matching substring in R
long vc=0,vcc=0,vcmatch=0; // offset of the first character of the current string
in V
long vm = 0;  // starting offset of a matching substring in V

int ln=0,ln2=0;                   //length of the longest matching string
int len=0,leng=0, len2=0;   //holds the length of the longest matching string
int longueur = 0;
int lngth = 0;

int bestoffsetR=0;
int bestoffsetV=0;

int matchingV = 0;
int matchingR = 0;

int ADDCOUNTER = 0;
int COPYCOUNTER = 0;


/*
..............................                    ..........................................
                                   END OF DECLARATIONS
..............................                    ..........................................

*/
```