Differencing Data Streams[*]

Sudarshan S. Chawathe

Department of Computer Science
University of Maryland
College Park, Maryland 20742, USA
chaw@cs.umd.edu

Abstract

We present external-memory algorithms for differencing large hierarchical datasets. Our methods are especially suited to streaming data with bounded differences. For input sizes m and n and maximum output (difference) size e, the I/O, RAM, and CPU costs of our algorithm rdiff are, respectively, $m + n$, $4e + 8$, and $O(MN)$. That is, given $4e + 8$ blocks of RAM, our algorithm performs no I/O operations other than those required to read both inputs. We also present a variant of the algorithm that uses only four blocks of RAM, with I/O cost $8me + 18m + n + 6e + 5$ and CPU cost $O(MN)$.

1 Introduction

We study the problem detecting differences between two related datasets. This *differencing problem* has **applications** in diverse areas, including view maintenance, data warehousing, change management, version control, and efficient screen updates [5, 10, 4, 18]. Typically, database systems manage the evolution of a database by requiring that changes to the database be made only through the database system. Triggers, integrity constraint management, and other active-database features may then be hooked into the update routines. However, in a heterogeneous environment composed of multiple autonomous agents a database is often updated outside the control of a database system. For example, consider a large XML dataset such as the Protein Data Base [24]. Copies of this dataset, controlled by a distributed group of researchers, may diverge as updates are made locally. In order to merge these changes back into the main repository, while detecting potential conflicts, we need a method that examines different versions of the dataset and computes the differences. In addition to merging changes, such differencing also forms the basis of more sophisticated database techniques [7]. For example, the changes, once detected, may be used to trigger notifications or other actions. Storing a history of changes permits the changes to be queried as well.

Our focus in this paper is on external memory and *streaming environments*, which we define as those that strongly favor sequential data access. Non-streaming access may be either unavailable or significantly more expensive. As an example, consider a VRML [2] dataset storing traces of a simulated flight for training or testing. For long and detailed simulations (such as those designed for rendering in a graphics cave), the VRML trace files can grow very large. It is often useful to compare two or more related traces. For example, differences between two similar test flights may reveal design problems that amplify minor pilot errors. In such applications, the data is so large and unwieldy that often streaming it is the only practicable access method. In general, streaming algorithms are useful not only when the data occur natively in a streaming form but also when streaming is the preferred mode of access for efficiency reasons.

Prior work has addressed several variants of the differencing problem. The key **distinguishing features** of the work presented in this paper are the following. (A detailed discussion of related work appears in Section 5.) 1. We study differencing techniques for datasets that are too large to fit in main memory. We develop external memory algorithms and analyze not only their dominant I/O cost, but also the required main-memory (RAM) space and the CPU cost. 2. We consider streaming environments in which data access is mainly through a sequential-read interface. To the best of our knowledge, there has been no prior work on differencing in streaming environments. 3. We study the differencing problems for both sequence and hierarchical (tree) data. While there is a substantial body of work on efficient algorithms for differencing sequences, algorithms for differencing trees are typically very expensive, even in main memory. (Indeed, several versions of the tree differencing

[*]This work was supported in part by the U.S. National Science Foundation with grants IIS-9984296, IIS-0081860, and CNS-0426683.

problem are NP-hard. We formulate a simpler version of the differencing problem for trees and present efficient algorithms for both main and external memory.)

The general strategy for **formulating a differencing problem** is the following: Data are *modeled* using a suitable structure (e.g, sequences, trees, graphs). A set of *edit operations* for transforming such data is defined. A sequence of edit operations is called an *edit script*; it transforms data by applying its operations in sequence. A cost model is chosen in order to associate a cost with each edit script. Finally, the differencing problem is defined to be the problem of finding a *minimum-cost edit script* that transforms one input to the other. Intuitively, an edit script describes the differences between two datasets by specifying how one is transformed to the other. There are typically infinitely many such edit scripts for a given pair of datasets. The minimum-cost requirement formalizes the idea that edit scripts making the fewest changes are desirable.

In this paper, we study two problem formulations. The first models **sequences** with only unit-cost insert and delete operations. We also study its natural extension to trees. The second formulation is more general. Data are modeled using **rooted, ordered, labeled trees**. Edit operations may insert, delete, or update a tree node and may have arbitrary costs (perhaps data dependent) associated with them.

The main **contributions** of this paper may be summarized as follows: 1. We present an external memory algorithm for differencing sequence data. To our knowledge, our algorithm is the first to apply to disk-resident data the main-memory technique of using diagonals in an edit graph to structure computation efficiently. 2. We present an extremely efficient algorithm for differencing hierarchical data in a streaming environment. Given a modest amount of main memory, our algorithm performs no I/O operations other than those required to read the streaming inputs sequentially. Given input sizes M and N, output (edit script) size E, and block size S, with $m = M/S$, $n = N/S$, and $e = E/S$, our algorithm performs $m + n$ I/O operations and uses $4e + 8$ blocks of main memory. Since the expected value of e is small for many applications (while m and n are quite large) a main-memory requirement that depends only on e and is independent of m and n is very useful. 3. We also present a variant of this algorithm for cases when sufficient main memory is unavailable. This variant requires only 4 blocks of main memory and performs $8me + 18m + n + 6e + 5$ I/O operations. Again, the small expected value of e makes the dominant term $8me$ favorable in many applications. The CPU cost of both variants is $O(MN)$.

Outline of the paper: We begin by formalizing our problem definitions and covering other preliminaries in Section 2. In Section 3, we present an external memory algorithm for comparing sequences. We describe how this algorithm uses edit graphs and diagonal-based computation and discuss extending it for trees. Our main algorithm for differencing streaming hierarchical data is presented in Section 4. We address related work in Section 5 and conclude in Section 6.

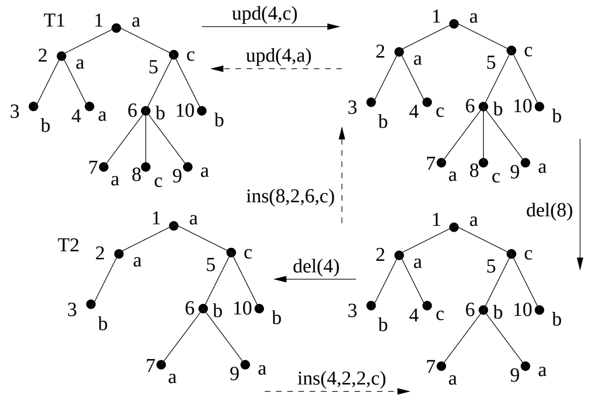

Figure 1. Edit operations on trees

2 Model and Problem Statement

We consider *rooted, ordered, labeled trees*. Each node in such a tree has an associated label. The children of a node are totally ordered. Thus, if a node has k children, we can uniquely identify the i'th child, for $i = 1 \dots k$. There is a distinguished node called the root of the tree. Formally, a rooted, ordered, labeled tree consists of a finite, nonempty set of nodes T and a labeling function l such that: (1) The set T contains a distinguished node r, called the **root** of the tree; (2) The set $V - \{r\}$ is partitioned into k disjoint sets $T_1, \dots, T_k$, where each T_i is a tree (called the ***i*'th subtree** of T or r); and (3) the label of a node $n \in T$ is $l(n)$ [15]. The root c_i of T_i is called the ***i*'th child** of the node r, and r is called the **parent** of c_i. Nodes in T that do not have any children are called **leaf nodes**; the rest of the nodes are called **interior nodes**.

Henceforth, we use the term trees to mean rooted, ordered, labeled trees. Some examples are depicted in Figure 1. Each node has an identifier depicted to its left and a label depicted to its right. The identifiers are for notational convenience only and are not part of the input. In particular, the identifiers cannot be used to match nodes in one tree with those in the other. For example, when we are comparing two VRML datasets, the node identifiers may represent offsets within the files or streams. Since the files are separate and may use different low-level formats, there is no correspondence between these offsets. Computing a correspondence between the nodes of the input trees is in fact the crux of the differencing problem.

We use the following **tree edit operations** to model and describe changes in trees:

Insertion: Let p be a node in a tree T, and let $T_1, \ldots, T_k$ be the subtrees of p. Let n be a node not in T, let l be an arbitrary label, and let $i \in [1, k+1]$. The insertion operation *ins*(n, i, p, l) inserts the node n as the i'th child of p. In the transformed tree, n is a leaf node with label l.

Deletion: Let n be a leaf node in T. The deletion operation *del*(n) results in removing the node n from T. That is, if n is the i'th child of a node $p \in T$ with children $c_1, \ldots, c_k$ then, in the transformed tree, p has children $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_k$.

Update: If n is a node in T and v is a label then the label-update operation *upd*(n, v) results in a tree T' that is identical to T except that, in T', $l(n) = v$.
We assume, without loss of generality, that the root of a tree is not inserted or deleted.

An **edit script** is a sequence of edit operations. The result of applying an edit script to a tree T is the tree obtained by applying the component edit operations to T, in the order they appear in the script. The solid arrows in Figure 1 suggest the application of an edit script that transforms the tree T_1 in the top left corner to the tree T_2 in the bottom left corner. The edit script is the concatenation of the edit operations on the arrows: *upd*$(4, c)$, *del*(8), *del*(4). The dashed arrows suggest the application of an edit script that transforms T_2 back to T_1.

In order to define a **cost model** for edit operations, let $c_i(x)$ and $c_d(x)$ denote the costs of, respectively, inserting and deleting node x. Let the cost of updating a label l_1 to l_2 be $c_u(l_1, l_2)$. The cost of an edit script is the sum of the costs of its component operations. We may now formally define the problem of differencing trees as follows:
Problem Statement (Trees): Given two rooted, labeled, ordered trees A and B, find a minimum-cost edit script that transforms A to a tree that is isomorphic to B.

A sequence is a special case of such a tree; that is, every sequence corresponds to a rooted, ordered, labeled tree of height 1. We may therefore define the problem of differencing sequences (or strings) as follows:
Problem Statement (Sequences): Given two sequences A B, find a minimum-cost edit script that transforms A to a sequence that is isomorphic to B.

3 A Sequence Comparison Algorithm

The problem of differencing data using even simple edit models is known to have a quadratic lower bound on time complexity in terms of input size [1, 23]. However, the expected-case running time can be improved by designing algorithms that optimize the expected common cases. In this section, we extend Myers's sequence differencing algorithm [13] to external memory, discuss its performance characteristics, and explore further extensions to hierarchical and stream data.

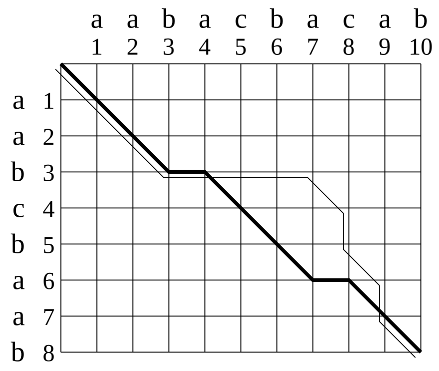

Figure 2. Edit graph for sequence comparison

The input to Myers's algorithm consists of two sequences of elements. The only permissible edit operations are element insertion and deletion, which are both assigned unit costs. The algorithm is based on two key ideas: First, the min-cost edit-script problem in this simple model is reduced to a shortest-path problem in a grid called the **edit graph**. Second, the shortest-path problem is solved efficiently by computing only those shortest paths that are close to the diagonal of the grid; we call this strategy **diagonal-based** computation. These ideas are elaborated below.

3.1 Edit Graphs

The **edit graph of two sequences** $A = (A[1]\ A[2] \ldots A[m])$ and $B = (B[1]\ B[2] \ldots B[n])$ is the $(m+1) \times (n+1)$ grid suggested by Figure 2. (Each point where two lines touch or cross is a node in the edit graph.) A point (x, y) intuitively corresponds to the pair $(A[x], B[y])$, for $x \in [1, m]$ and $y \in [1, n]$. In our edit graphs, the origin $(0, 0)$ is the node in the top left corner; the x-axis extends to the right of $(0, 0)$ and the y-axis extends down from $(0, 0)$. There is a directed edge from each node to the node, if any, to its right. Similarly, there is a directed edge from each node to the node, if any, below it. For clarity, these directed edges are shown without arrowheads in the figure. All horizontal edges are directed to the right and all vertical edges are directed down. In addition, there is a diagonal edge from $(x-1, y-1)$ to (x, y) for all $x, y > 0$. For clarity, these edges are omitted in the figure. The edit graph depicted in Figure 2 corresponds to the sequences (strings) $A = ababaccdadab$ and $B = acabbdbbabc$.

Traversing a horizontal edge $((x-1, y), (x, y))$ in the edit graph corresponds to deleting $A[x]$. Similarly, traversing a vertical edge $((x, y-1), (x, y))$ corresponds to inserting $B[y]$. Traversing a diagonal edge $((x-1, y-1), (x, y))$

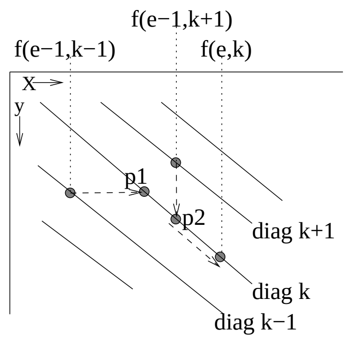

Figure 3. Computing with diagonals

corresponds to matching $A[x]$ to $B[y]$; if $A[x]$ and $B[y]$ differ, such matching corresponds to an update operation. Edges in the edit graph have **weights** equal to the costs of the edit operations they represent. Thus, a horizontal edge $((x-1,y),(x,y))$ has weight $c_d(a_x)$, a vertical edge $((x,y-1),(x,y))$ has weight $c_i(b_y)$, and a diagonal edge $((x-1,y-1),(x,y))$ has weight $c_u(a_x,b_y)$. The **weight of a path** is the sum of the weights of its constituent edges. It is easy to show that any min-cost edit script that transforms A to B can be mapped to a path from $(0,0)$ to (M,N) in the edit graph. Conversely, every path from $(0,0)$ to (M,N) corresponds to an edit script that transforms A to B [13].

Figure 2 suggests two paths from source to sink. Consider first the path depicted using the thinner line. This path follows four horizontal edges and two vertical edges. The delete operations denoted by the horizontal edges are obtained by reading the identifier of the node above the target of each horizontal edge, while the inserted nodes are read from the left of the targets of vertical edges. Following this procedure yields *del*$(A[4])$, *del*$(A[5])$, *del*$(A[6])$, *del*$(A[7])$, *ins*$(B[5])$, *ins*$(B[7])$ as the edit script represented by the thinner path. We can verify that this edit script does indeed transform A to B. However, it is not a minimum-cost edit script. The edit script corresponding to the bold path in Figure 2, *del*$(A[4])$, *del*$(A[8])$, has a lower cost.

3.2 Using Diagonals

A standard method for shortest path (Dijkstra's algorithm) requires $O(E \log V)$ time in a graph with E edges and V vertices [6]. However, the $\log V$ term can be removed using special implementations of the priority queue used in the shortest-path algorithm. Further, due to the structure of the edit graph, E and V are both $O(MN)$, yielding an $O(MN)$ algorithm. The performance can be further improved by considering the special grid structure of edit graphs.

We will refer to the set of edit graph points $\{(x,y) : x - y = k\}$ as the **k'th diagonal**. Thus, the source $(0,0)$ is on the 0'th diagonal and the sink (M,N) is on the $(M-N)$'th diagonal. Consider the band of diagonals consisting of the 0'th diagonal, along with e diagonals above it and e below it. We call this set of diagonals the e**-band**. Any path from the source to a point outside the e-band must traverse at least $e+1$ horizontal or vertical edges and must therefore cost at least $e+1$. It follows that there are at most $\min(M,N)\cdot D$ points in the edit graph at a distance D or less from the source. Since the single-source shortest-path algorithm explores vertices in order of increasing distance from the source and stops once the sink is reached, it takes only $O((M+N)D)$ time. (The output parameter D is the cost of a min-cost edit script, i.e., the distance of the sink from the source.)

Myers's algorithm uses the idea of diagonals to compute the shortest-path distances directly without using Dijkstra's algorithm. Consider the furthest point (in terms of its x and y coordinates) on diagonal k that is reachable from the source using a path of weight e; this point is called the **furthest-reaching** e**-point on diagonal** k and the path is called the **furthest-reaching** e**-path on diagonal** k. The furthest-reaching e-path on diagonal k can always be decomposed into the furthest-reaching $(e-1)$-path on either diagonal $k-1$ or $k+1$, followed by a horizontal or vertical edge, respectively, followed by a 0-weight path (if any) along diagonal k [13]. This **path decomposition property** is what enables the diagonals-based method of computing distances: We can compute the furthest-reaching e points on diagonals $-e,\ldots,e$ recursively from the furthest-reaching $e-1$ points on diagonals $-e+1,\ldots,e-1$.

Figure 3 is a graphical illustration of such a computation. To obtain the furthest-reaching e point on diagonal k, we first find the projections p_1 and p_2 of the furthest-reaching $e-1$ points on diagonals $k-1$ and $k+1$. We then pick the point p_i that is furthest from the origin and extend the path along the k'th diagonal by traversing diagonal edges until we are at a point with no diagonal out-edge. This point is the furthest e point on diagonal k. In Figure 3, p_2 is further than p_1, so we traverse diagonal k starting at p_2. More precisely, we have the following recurrence for $f(e,k)$, the x-coordinate of the furthest e-point on diagonal k. (The y-coordinate follows as $f(e,k)-k$.)

$$
\begin{aligned}
f(e,k) \quad = \quad & s(f(e-1,k+1),k) && (1)\\
& \qquad \text{if } k=-e \text{ or}\\
& \qquad f(e-1,k-1) < f(e-1,k+1)\\
& s(f(e-1,k-1)+1,k)\\
& \qquad \text{otherwise}
\end{aligned}
$$

where $s(x)$ intuitively represents the operation of traversing a maximal path of zero-weight diagonal edges from $(x, x-k)$. That is, $s(x,k) = (x+i, x-k+i)$ such that $\forall j \in$

$[0, i-1] : c_u([(x+j, x-k+j), (x+j+1, x-k+j+1)]) = 0$ and $c_u([(x+i, x-k+i), (x+i+1, x-k+i+1)] \neq 0$.

The above recursion immediately suggests a main memory algorithm that starts by computing the furthest 0-point on diagonal 0 and at each stage uses the furthest k-points on the current set of diagonals to compute the furthest $k+1$-points on their neighboring diagonals. Below, we present an external memory algorithm that builds on this main memory algorithm.

3.3 Algorithm L1

Input: Sequences A and B stored on disk.
Output: The cost of a minimum-cost edit script that transforms A to B using unit cost insert and delete operations, if the cost is no greater than D_m, and -1 otherwise.
Method: Figure 4 lists the pseudocode for Algorithm L1. The algorithm computes the furthest e-points on diagonals $-e, \ldots, e$ for increasing values of e until the sink of the edit graph is reached or until e exceeds the limit D_m. The outermost nested for loops set up this scheme. We use RdBlk$(X, \text{'F'}, b)$ to denote reading block b of file F into buffer X. Similarly, we use WrBlk$(X, \text{'F'}, b)$ to denote writing the contents of buffer X to block b of file F. The main data structure is an array V containing the x-coordinates of the furthest k-points computed so far. Immediately before the inner for loop, $V[i]$ is the furthest $(k-1)$-point on on diagonal i, for $i = -k+1, \ldots, k-1$. Immediately after that loop, $V[i]$ is the furthest k-point on diagonal i for $i = -k, \ldots, k$. The if statement on line 10 ensures that for each iteration of the outer loop, the inner for loop updates the V array for only every other diagonal, starting with $-k$; the other values are simply copied (line 36). The new values are computed using the neighboring diagonals based on Equation 1. The V array is updated in place by using temporary variables v' and v'' which hold the values of $V'[k_o - 1]$ and $V'[k_o - 2]$ from the previous iteration of the outer for loop.

Figure 5 depicts the relationship between the V-arrays of successive iterations. The arrows indicate the flow of information from the array cells of V in one iteration to those in the next. Only every other cell in V is updated, based on the values of cells representing the furthest d-points on the diagonals to either side of the current cell's diagonal. The remaining cells are copied over. Since the V array grows by two cells from one iteration to the next, the array indexes are shifted by one position at each iteration. This observation explains the use of $V[k_o]$ instead of $V[k_o - 1]$ in lines 11 and 12 of Figure 4. The test on line 33 checks whether the traversal has reached the sink of the edit graph; if so, the current value of e is the cost of a shortest path from source to sink, i.e., the cost of a min-cost edit script from A to B. If the loops terminate before this condition is met, it means there is no path of cost D_m or less, and we signal this fact by returning a special value. The rest of the pseudocode in Figure 4 is responsible for transferring the the required parts of arrays V, A, and B between disk and RAM. The array index x is mapped to a block address x_b and block offset x_o on line 17. If the required block x_b is not the block x'_b currently in buffer A then block x_b is read in on lines 21–24. The buffers B and V are handled analogously on lines 4–8 and 25–28, respectively.

Performance: Algorithm L1 has very low RAM costs, requiring space for only the three buffers A, B, and V in addition to scalar variables. The CPU cost is dominated by the while loop on line 16, which traverses diagonals in the edit graph. Since no node in the graph is visited more than once in this manner and since at most D diagonals are traversed (where D is the optimal edit distance), the cost of this part of the code is $O(D(M+N))$. At iteration e of the outer for loop, the V array has $2e+3$ elements and occupies $\lceil (2e+3)/S \rceil$ blocks (where S is the block size), each of which is read and written back (ignoring the termination case). The outer for loop iterates through $e = 0, \ldots, D$. Thus the number of I/Os incurred in accessing V is $(d+1)(d+2)+3d$ (ignoring boundary cases, where $d = D/S$). Note that d is expected to be small and the V array is accessed sequentially. The order of accesses to the files A and B is not fixed, being data dependent. When diagonal k is traversed from $(x, x-k)$ to $(x+g, x+g-k)$, ranges $A[x..x+g]$ and $B[x+g..x+g-k]$ of the input files must be read. Unfortunately, the same portion of input file A may be read multiple times, since different diagonals may traverse that range in different (widely separated) iterations of the outer for loop.

3.4 Extension to Trees

The idea of mapping the sequence comparison problem to a shortest path problem in an edit graph can be extended to our model for editing trees. Intuitively, the structure of a tree imposes constraints on the manner in which nodes may be inserted and deleted. For example, the deletion of an interior node implies the deletion of its descendants. Such constraints can also be phrased in terms of disallowing certain paths in the edit graph. However, explicitly testing such path constraints is too expensive and negates the performance benefits of using an edit graph. Fortunately, we can define a **tree edit graph** that incorporates all needed constraints implicitly in its structure.

The edit graph for sequence comparison is a complete grid with both horizontal and vertical edges out of all but the boundary vertices. In contrast, the vertices in a tree edit graph may be missing a horizontal or vertical outedge. Such missing edges encode the information that cer-

edge type	edge	range	constraint
diagonal	$[(x,y),(x+1,y+1)]$	$0 \le x < M, 0 \le y < N$	$A[x+1].d = B[y+1].d$
horizontal	$[(x,y),(x+1,y)]$	$0 \le x < M, 0 \le y \le N$	$y = N$ or $B[y+1].d \le A[x].d$
vertical	$[(x,y),(x,y+1)]$	$0 \le x \le M, 0 \le y < N$	$x = M$ or $A[x+1].d \le B[y].d$

Figure 6. Edge constraints for a tree edit graph

```
(0)  $(x'_b, y'_b) \leftarrow (-1,-1)$;
(1)  for $e \leftarrow 0$ to $D_m$ do begin
(2)    for $k \leftarrow -e$ to $e$ do begin
(3)      $(k_b, k_o) \leftarrow ((e+k)$ div $S, (e+k)$ mod $S)$;
(4)      if $k_o = 0$ then begin
(5)        if $k_b > 0$ then WrBlk($V$, 'V', $k_b - 1$);
(6)        RdBlk($V$, 'V', $k_b$);
(7)        if $k = 0$ then $V[0] \leftarrow 1$;
(8)      end;
(9)      $(v'', v') \leftarrow (v', V[k_o])$;
(10)     if $(e+k)$ mod $2 = 0$ then begin
(11)       if $k = -e$ or $k \ne e$ and $v'' < V[k_o]$ then
(12)         $x \leftarrow V[k_o]$;
(13)       else
(14)         $x \leftarrow v'' + 1$;
(15)       $y \leftarrow x - k$;
(16)       while $x < M$ and $y < N$ do begin
(17)         $(x_b, x_o) \leftarrow ((x+1)$ div $S, (x+1)$ mod $S)$;
(18)         $(y_b, y_o) \leftarrow ((y+1)$ div $S, (y+1)$ mod $S)$;
(19)         if $x_b \ne x'_b$ then
(20)         $(y_b, y_o) \leftarrow ((y+1)$ div $S, (y+1)$ mod $S)$;
(21)         if $x_b \ne x'_b$ then begin
(22)           RdBlk($A$, 'A', $x_b$);
(23)           $x'_b \leftarrow x_b$;
(24)         end;
(25)         if $y_b \ne y'_b$ then begin
(26)           RdBlk($B$, 'B', $y_b$);
(27)           $y'_b \leftarrow y_b$;
(28)         end;
(29)         if $A[x_o] \ne B[y_o]$ then break;
(20)         $(x,y) \leftarrow (x+1, y+1)$;
(31)       end;
(32)       $V[k_o] \leftarrow x$;
(33)       if $x \ge M$ and $y \ge N$ then return($e$);
(34)     end;
(35)     else
(36)       $V[k_o] \leftarrow v'$;
(37)   end;
(38) end;
(39) return($-1$);
```

Figure 4. Algorithm L1

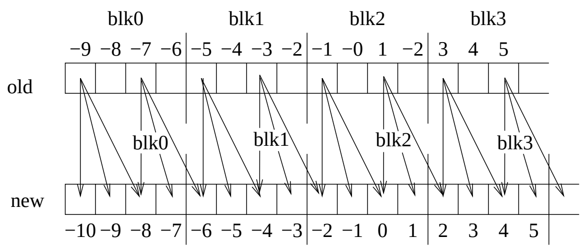

Figure 5. Updating the V array in Algorithm L1

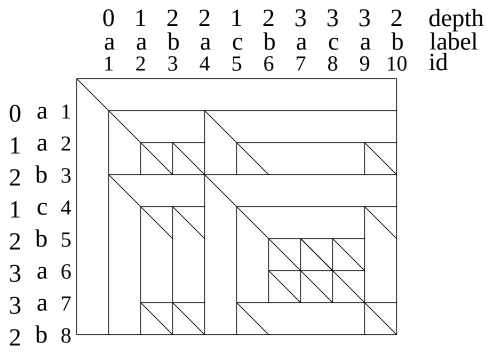

Figure 7. Edit Graph for Trees

tain paths are not permissible. Tree edit graphs have two key features. First, the missing edges disallow exactly those paths that correspond do invalid edit scripts. Second, the presence of edges is encoded using a simple arithmetic test.

Tree edit graphs use a tree representation based on listing the label and depth of each node: We define the **ld-pair** of a node with label l and depth d to be the pair (l, d). The **ld-pair representation of a tree** is simply the preorder listing of the ld-pairs of its nodes. A tree with M nodes is thus represented as an array $A[1..M]$ where $A[i]$ is the ld-pair of the i'th node in preorder. We use the notation $A[i].l$ and $A[i].d$ to denote the components of the ld-pair $A[i]$. Henceforth, we shall assume that the input trees are in this format.

Given trees $A[1..M]$ and $B[1..N]$, their edit graph consists of grid of nodes $\{(x,y) : 0 \le x \le M, 0 \le y \le M\}$ with the edges indicated in Figure 6.

A path from the source $(0,0)$ to a point (x,y) in the edit graph must pass through one the nodes $(x-1,y)$, $(x,y-1)$, and $(x-1, y-1)$ because these are the only nodes with edges leading to (x, y). It follows that the distance of (x, y) (from the source) is the minimum of the distances of these three points from the source plus the cost of the connecting edge. One or more (but not all three) of the potential in-edges to (x, y) may be missing, so we must test for edge presence. These observations lead to the following recurrence for $D(x, y)$, the distance of (x, y) from the origin, where G denotes the edit graph:

$$\begin{aligned}
D(x,y) &= \min\{m_1, m_2, m_3\} \quad \text{where} \qquad (2)\\
m_1 &= D(x-1,y-1) + c_u(A[x], B[y]) \\
&\quad\ \text{if } ((x-1,y-1),(x,y)) \in G \\
&\quad \infty \quad \text{otherwise} \\
m_2 &= D(x-1,y) + c_d(A[x]) \\
&\quad\ \text{if } ((x-1,y),(x,y)) \in G \\
&\quad \infty \quad \text{otherwise} \\
m_3 &= D(x,y-1) + c_i(B[y]) \\
&\quad\ \text{if } ((x,y-1),(x,y)) \in G \\
&\quad \infty \quad \text{otherwise}
\end{aligned}$$

The edge presence tests are easily implemented using the arithmetic tests described above.

Unfortunately, Algorithm L1 cannot be applied in a straightforward manner to tree edit graphs because the path decomposition property described in Section 3.2 does not hold for tree edit graphs. In effect, there may be no horizontal edge connecting the furthest reaching e-point on diagonal $k-1$ to a point on diagonal k. Similarly, the desired vertical edge connecting the furthest-reaching e-point on diagonal $k+1$ to diagonal k may be missing. It is possible to formulate an alternate path decomposition property for tree edit graphs that works around such problems. However, the result is not as clean as the property for sequence edit graphs. Further, even with such a workaround, we are left with the performance problems described in Section 3.3 when applying the results to external memory. Therefore, we do not pursue this strategy further in this paper. Instead, in the next section we present an efficient streaming algorithm that uses the ideas of this section without relying on the path decomposition property.

4 Differencing Streaming Trees

In this section, we present our algorithm rdiff which uses ideas from the previous section to efficiently difference streaming data. Intuitively, the problem with the diagonal-based computation in Algorithm L1 is that accesses to physically contiguous portions of the inputs are scattered over time, resulting in a block being read multiple times. This observation suggests a strategy of performing all the com-

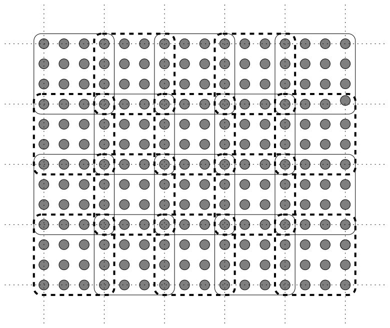

Figure 8. Tiling the distance matrix

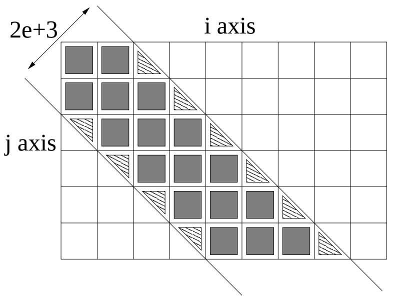

Figure 9. A D-band of distance matrix tiles

putations on an input block immediately after it is read. Following through on this strategy leads to the blocked version of the classic Wagner-Fisher algorithm [19, 3], with quadratic I/O cost. It seems as though the idea of computing distances using diagonals in the edit graph and the idea of performing all required computation on input data immediately after it is read are incompatible. However, it is possible to decouple the idea of computing distances using diagonals from the idea of restricting computation to diagonals in a certain range. We present the details below after covering a few preliminaries.

We define a **distance matrix** D to be the set $\{D(x, y) : 0 \leq x \leq M, 0 \leq y \leq N\}$ where, as in the previous section, $D(x, y)$ denotes the distance of the edit graph point (x, y) from the origin $(0, 0)$. Figure 8 represents this matrix pictorially using dots for entries. Our algorithm is based on **tiling** the distance matrix as suggested by the figure. Note that we are abusing the term tiling, since neighboring tiles actually overlap on their boundary elements.

Our algorithm is based on two key ideas: First, given the top and left edges of a tile, we can compute the distance submatrix for the rest of the tile by using the recurrence of Equation 2 in Section 3.4. Further, storing only the tile edges incurs significantly smaller I/O costs compared with storing the entire distance matrix. We refer to the set of distance-matrix entries that lie on tile edges as the **distance grid**. Since each tile edge contains S elements, all the mn tiles can be stored on disk using $2mnS$ units or $2mn$ blocks.

The second key idea behind our algorithm is that of restricting computation to a band of tiles close to the 0'th diagonal. Specifically, we do not compute distances for tiles that lie completely outside the band of $2D + 1$ diagonals centered at the origin. We call this region the **D-band**. (See Figure 9.)

Within the D-band, we organize our computation by traversing the tiles in the D-band in column-major order. Since the border of the D-band cuts diagonally across the tiles near its top edge, we are faced with the prospect of computing the distances in such a tile without access to its top edge (which is outside the D-band). This situation is resolved by assuming all distances on the top edge to be infinity for the purposes of computing the tile using Equation 2 in Section 3.4. Recall that every insert and delete operation costs at least one unit. Thus, a path cannot exit the D-band without exceeding D in cost. Therefore, setting the top edge distances to infinity does not introduce errors in the computation of distances within the D-band.

4.1 Algorithm rdiff

Input: Data streams S_A and S_B representing, in the l-d pair notation (Section 3.4), trees with M and N nodes, respectively.
Output: The cost of a minimum-cost edit script from tree S_A to tree S_B, if that cost is at most e; ∞ otherwise.
Method: Figure 10 lists the pseudocode for Algorithm rdiff. In the following description, we assume that the first block of input A has a dummy first node, $(0, 0)$. Each successive block has as its first node a copy of the last node from the previous block. The blocks of B are also assumed to be in this format. We also assume that the input sizes M and N are both integral multiples of $S' = (S - 1)$, where S is the block size, with $m = M/S$ and $n = N/S$. These assumptions are not necessary for our algorithm; we make them only to simplify its presentation. Using the nested for loops, the algorithm traverses all tiles in the e-band of the tiled edit graph. The indices i and j identify the (i, j)'th tile. The algorithm uses a buffer to hold the most recently read block of data from input stream S_A. Blocks from the input streams are read (sequentially) using the *getNextBuffer* function. For the input stream S_B, we store the last $2e + 3$ blocks read in buffers $B_{-e-1}, \ldots, B_{e+1}$. To avoid confusion between indices of buffers and indices within buffers, we use subscripts (e.g., B_2) for the former and brackets (e.g., $A[5]$) for the latter. The procedure *leftShiftBuffers* assigns the contents of buffer B_k to buffer B_{k-1} for $k = -e \ldots e + 1$, essentially maintaining a FIFO queue of the $2e + 3$ most recently read blocks of stream

```
(1)  $(a, b) \leftarrow (0, 0)$;
(2)  for $i \leftarrow 0$ to $m - 1$ do begin
(3)    $H \leftarrow (\infty, \ldots, \infty)$;
(4)    $A \leftarrow$ getNextBuffer($S_A$);
(5)    leftShiftBuffers($B$);
(6)    $B_{e+1} \leftarrow$ getNextBuffer($S_B$);
(7)    leftShiftBuffers($V$);
(8)    $V_{e+1} \leftarrow (\infty, \ldots, \infty)$;
(9)    for $j \leftarrow \max\{0, i - e - 1\}$ to $i + e + 1$ do begin
(10)     if $j = 0$ then initD('H', $H, A, a$);
(11)     if $i = 0$ then initD('V', $V_{j-i}, B_{j-i}, b$);
(12)     processTile($A, B_{j-i}, H, V_{j-i}$);
(13)   end;
(14)   if allInfinity($V$) then return $\infty$;
(15) end;
(16) return $H[S']$;
```

Figure 10. Algorithm rdiff

```
(1)  procedure initD($d, D, F, a$)
(2)    $D[0] \leftarrow a$;
(3)    for $i \leftarrow 1$ to $S'$ do begin
(4)      if $d$ = 'H' then $c \leftarrow c_d(F[i])$;
(5)      else $c \leftarrow c_i(F[i])$;
(6)      $D[i] \leftarrow D[i - 1] + c$;
(7)    end;
```

Figure 11. Procedure initD

S_B. We use leftShiftBuffers as described mainly as a notational convenience to simplify subscripting. Its implementation can avoid the in-memory data copy operations by using a pointer-based data structure. The buffer H holds the top (horizontal) edge of the current tile, while the buffers $V_{-e-1}, \ldots, V_{e+1}$ hold the left (vertical) edges of the tiles in the current (i'th) column of the e-band.

Let us ignore the two if statements in the inner loop for now; they handle the boundary cases discussed below. The bulk of the work for each tile is done by the procedure *processTile*, which is explained below. This procedure takes as inputs buffers (A and B_{j-i}) containing segments of the two inputs corresponding to the current tile (i, j), along with the buffers holding the top and left edges (H and V_{j-i}). When processTile procedure returns, H and V_{j-i} have been updated in-place (destructively) to hold the bottom and right edges of the current tile.

The algorithm thus uses the top and left edges of each tile in the e-band to compute its bottom and right edges. When the loops terminate, the buffer H contains the lower edge of the tile in the bottom right corner of the edit graph. Its last entry is the required cost of a min-cost path from

```
(1)  procedure processTile$(A, B, H, V)$
(2)    $H[0] \leftarrow V[S']$;
(3)    for $i \leftarrow 1$ to $S'$ do begin
(4)      $t \leftarrow V[0]$;
(5)      $V[0] \leftarrow H[i]$;
(6)      for $j \leftarrow 1$ to $S'$ do begin
(7)        $(m_1, m_2, m_3) \leftarrow (\infty, \infty, \infty)$;
(8)        if $A[i].d = B[j].d$ then $m_1 \leftarrow t + c_u(A[i], B[j])$;
(10)       if $j = S'$ or $A[i].d \geq B[j+1].d$
             then $m_2 \leftarrow V[j] + c_d(A[i])$;
(11)       if $i = S'$ or $A[i+1].d \leq B[j].d$
             then $m_3 \leftarrow V[j-1] + c_i(B[j])$;
(12)       $t \leftarrow V[j]$;
(13)       $V[j] \leftarrow \min\{m_1, m_2, m_3\}$;
(14)     end;
(15)     $H[i] \leftarrow V[S']$;
(16)   end;
```

Figure 12. Procedure processTile

the source to the sink, that is, the cost of a min-cost edit script between the two inputs. The statement with a call to function *allInfinity* is an optimization that terminates the algorithm early if all entries in all V buffers are found to be ∞. (In such a case, we know that the sink cannot be reached by a path from the source that stays completely within the e-band.)

Procedure processTile: This procedure computes all distances in the distance matrix tile corresponding to its inputs using the nested loops to iterate over all positions in the tile. The distances (costs) are computed based on Equation 2 in Section 3.4. The entire distance submatrix of the tile is not materialized at once (to avoid $O(S^2)$ RAM cost); instead, the H and V buffers are modified in-place to progress stepwise to the bottom and right, respectively.

Boundary cases: The above description holds for the bulk of the processing done by the the algorithm, when the indices i and j are not close to the edges of the distance matrix or the e-band. Let us now consider the boundary cases.

The first boundary case is when we are at a tile that is at the upper limit of the e-band. For such a tile, the top edge has not been previously computed, since the tile that would perform this computation lies above the e-band. However, we know that the distances in this edge must be greater than e and are thus ∞ for our purposes. This case is therefore handled by initializing all positions in H to ∞ by default. The symmetric boundary case is when we are at a tile that is at the lower limit of the e-band. For such a tile, the left edge has not been previously computed, since the tile that would perform this computation lies below the e-band. However, as above, we know that the distances in this edge must be greater than e and are thus ∞ for our purposes. These two boundary cases are handled by lines 3 and 8 in Figure 10.

The next boundary case is when we are at a tile on the upper boundary of the distance matrix; i.e., when $j = 0$. For such a tile, the top edge has not yet been computed, since there is no tile above it. From the structure of the edit graph, we know that a path can reach a point on the upper boundary of the edit graph only if it follows only horizontal edges from the source to that point. Therefore, this case is handled (on line 10 of Figure 10) by computing costs along this upper boundary. We use an accumulator variable a for this purpose, adding the edge costs in procedure *initD*. The symmetric case of a tile on the left boundary of the distance matrix is handled analogously on line 11.

Performance: Algorithm rdiff performs no I/O other than reading the two inputs sequentially using the getNextBuffer function; thus, the I/O cost is $m + n$. Other than the small amount of RAM space required for scalar variables and program state, the algorithm requires space for the buffers A, $B_{-e-1}, \ldots, B_{e+1}$, H, and $V_{-e-1}, \ldots, V_{e+1}$: $1 + (2e+3) + 1 + (2e+3) = 4e + 8$ blocks. The CPU cost is easily seen to be $O(MN)$ since only constant-cost operations are performed within the nested loops that dominate the cost. We may summarize Algorithm rdiff's performance as follows:

I/O	RAM	CPU
$m + n$	$4e + 8$	$O(MN)$

4.2 Recovering the Edit Script

As presented the above algorithm returns only the cost of a min-cost edit script, not the script itself. Fortunately, it is easily modified to return the script. Before destructively modifying the buffers containing distances of points on tile edges, we write the old contents of these buffers out to disk. When the algorithm terminates successfully, we will have on disk the bottom and right edges of each tile in the D-band, indexed by the tile identifiers (i, j) (along with the top and left edges of those D-band tiles that are also on the top and left boundaries of the distance matrix). We call this set of distance matrix edges the **D-band grid**.

More precisely, we modify Algorithm rdiff by inserting the following lines at positions suggested by their numbers relative to the line numbers in Figure 10. The last two lines below write the bottom and right edges of a tile to disk immediately after they are computed. The first two lines write out the edges on the top and left boundaries of the edit graph.

```
(10.5)  if $j = 0$ then putStreamBuffer$(D, H)$;
(11.5)  if $i = 0$ then putStreamBuffer$(D, V_{j-i})$;
```

```
(12.5)  putStreamBuffer($(D, H)$);
(12.6)  putStreamBuffer($(D, V_{j-i})$);
```

Note that since the computation follows a regular pattern (down the D-band in column-major order), we can write the buffers H and V to a stream output and maintain associative lookup based on tile (i, j). Thus these disk operations proceed at sequential-access speeds.

Given the D-band grid on disk, a minimum-cost edit script is recovered as follows We start at the sink, (M, N), and traverse an optimal path backwards by using the recurrence in Equation 2 (Section 3.4) backwards. The only difference is that instead of direct access to all needed distance matrix points, we must now recompute the distance matrix points in each tile using its top and left edges (as is done in procedure processTile). During this recovery procedure, each block of the inputs S_A and S_B is accessed exactly once, giving $m + n$ I/Os. Note that these reads of the input data are sequential in reverse order. In moving from one distance matrix tile to the next, we move to either the left, the top, or the top-left direction, bringing us at least one tile closer to the origin. Since the longest path in the edit graph has $m+n$ tiles, it follows that at most $m+n$ tiles are recomputed, with an I/O cost of $2(m + n)$ for reading in the top and left edge of each tile. These reads, while not sequential, are still in reverse physical order and can thus be efficiently implemented in a stream interface that supports skipping regions of the input. The total I/O cost of the recovery is thus $3(m + n)$. Although CPU operations are quadratic in the block size (length of a tile edge), the block size is constant and thus the CPU cost is $O(M + N)$. The RAM costs of the recovery method as described is $4S + S^2$; however, using techniques similar to those in [3], the RAM cost can be reduced to $6S$.

4.3 Reducing RAM requirements

Recall from Section 4.1 that Algorithm rdiff requires $4e + 8$ blocks of RAM buffer space, where e is the maximum (interesting) edit distance between the inputs. The maximum edit distance e is expressed in units of blocks; that is $e = \lceil E/S' \rceil$, where E is the maximum edit distance in scalar units. In applications of interest to us, such as detecting and marking up differences between two versions of a VRML file, e is expected to be small. However, even in such applications we may encounter cases when e is large and $4e + 8$ blocks of RAM are unavailable. Fortunately, a few modifications to Algorithm rdiff permit it to run with very low RAM requirements, with a modest increase in I/O cost. We describe the modified algorithm, called **rdiff2**, below.

Instead of storing the buffer arrays $B_{-e-1}, \ldots, B_{e+1}$ and $V_{-e-1}, \ldots, V_{e+1}$ in RAM, we use $2e+3$ blocks on disk for each buffer array. The disk blocks for each buffer array are organized as a simple FIFO queue. We assume a stream interface to this queue; however, this assumption is not critical since we can emulate the stream interface using the standard disk interface. In particular, we use enQStr(S, X) to denote the operation that enqueues buffer X in queue S, and $X \leftarrow$ deQStr(S) to denote the dequeuing operation.

```
(1)  $(a, b) \leftarrow (0, 0)$;
(2)  $B \leftarrow (0, \ldots, 0)$;
(3)  $V \leftarrow (0, \ldots, 0)$;
(4)  for $j \leftarrow -e-1$ to $e+1$ do begin
(5)    if $j \geq 0$ then $B \leftarrow$ getNextBuffer$(S_B)$;
(6)    enQStr$(S'_B, B)$; enQStr$(S'_V, V)$;
(7)  end;
(8)  for $i \leftarrow 0$ to $m-1$ do begin
(9)    $H \leftarrow (\infty, \ldots, \infty)$;
(10)   $A \leftarrow$ getNextBuffer$(S_A)$;
(11)   $V \leftarrow$ deQStr$(S'_V)$;
(12)   $V \leftarrow (\infty, \ldots, \infty)$;
(13)   enQStr$(S'_V, V)$;
(14)   $B \leftarrow$ deQStr$(S'_B)$;
(15)   $B \leftarrow$ getNextBuffer$(S_B)$;
(16)   enQStr$(S'_B, B)$;
(17)   for $j \leftarrow i-e-1$ to $i+e+1$ do begin
(18)     $V \leftarrow$ deQStr$(S'_V)$;
(19)     $B \leftarrow$ deQStr$(S'_B)$;
(20)     if $j = 0$ then initD('H'$, H, A, a)$;
(21)     if $i = 0$ and $j \geq 0$ then initD('V'$, V, B, b)$;
(22)     if $j \geq 0$ then processTile$(A, B, H, V)$;
(23)     enQStr$(S'_B, B)$;
(24)     enQStr$(S'_V, V)$;
(25)   end;
(26)   if allInfinity$(V)$ then return $\infty$;
(27) end;
(28) return $H[S']$;
```

Figure 13. Algorithm rdiff2

Method: Figure 13 depicts the pseudocode for Algorithm rdiff2. (The inputs and outputs are identical to those of rdiff.) Most of the code is similar to that in Figure 10 and we focus on only the key differences here. Consider the steady state (ignoring boundary conditions for now) processing in the inner for loop (lines 18–24). The required buffers for B and V (corresponding to buffers B_{j-i} and V_{j-i} in Figure 10) are read in from their respective FIFO queues.on lines 18–19. The next three lines are completely analogous to lines 10–12 in Figure 10. The buffer B is enqueued again on line 23 since each block of the input stream S_B is used by $2e + 3$ columns of the D-band grid being computed. Lines 13–15 are responsible for changing the contents of the FIFO queue S'_B used as a temporary store for S_B's blocks. Before progressing to the next column of the D-band grid (next set of iterations of the inner loop), the earliest enqueued S'_B block is removed and the next block from S_B is enqueued. Line 24 inside the inner loop and lines 11-12 outside it perform a similar function for the V buffers. The rest of the new code handles the boundary conditions. Lines 2–7 prime the FIFO queues with sentinel elements to enable proper start-up conditions for the steady state code in the for loops. The use of sentinel elements incurs unnecessary I/O operations and can easily be avoided; we use it here since it simplifies the presentation.

Performance: Let us consider the additional I/O incurred in accessing the FIFO queues for the B and V buffers. Each iteration of the inner for loop incurs 4 additional I/O operations (lines 18, 19, 23, 24). Since there are $m(2e + 3)$ iterations, the total I/O cost from these lines is $4m(2e + 3)$. Similarly, the new I/O operations on lines 10, 11, 13, 14, and 16 contribute a total of $5m$ I/O operations. Finally, lines 4–7 contribute $(2e + 2) + (4e + 3) = 6e + 5$ operations. The total number of additional I/O operations is therefore $4m(2e+3)+5m+6e+5 = 8me+17m+6e+5$. Adding this number to the I/O cost $m + n$ of the original Algorithm rdiff gives $8me+18m+n+6e+5$ as the total I/O cost. This algorithm requires RAM storage for only the four buffers A, B, H, and V (in addition the small amount needed for scalar variables and program state). The CPU cost remains $O(MN)$ since only constant work is done within the nested loops. We may summarize Algorithm rdiff2's performance as follows:

I/O	RAM	CPU
$8me + 18m + n + 6e + 5$	4	$O(MN)$

5 Related Work

Main memory differencing algorithms have been extensively studied, especially for sequences [14]. One of the earliest algorithms for differencing strings (sequences) is the classic $O(mn)$ dynamic-programming algorithm due to Wagner and Fischer [19] (where m and n denote the sizes of the two inputs). For trees, Selkow's recursive algorithm uses an edit model similar to ours and has quadratic running time. A quadratic lower bound for sequence differencing (and therefore tree differencing in Selkow's model) is well known [1, 23]. For finite alphabets, the use of the four-Russians technique yields an $O(nm/\log n)$ algorithm [11].

Edit graphs for sequence comparison and diagonal-based shortest-path computations were introduced by Myers [13], along with a linear-space enhancement that forms the basis of the *diff* utility found on most current Unix systems [12]. Earlier implementations of Unix diff used the Hunt and Szymanski algorithm, with running time $O((R + N) \log N)$ where R is the number of ordered pairs of positions at which the two inputs match [9]. The parameter R is thus data-dependent, a property it shares with the parameter D in Myers's algorithm. However, while D is N in the worst case, R could be as large as N^2 yielding $O(N^2 log N)$ as the worst-case running time. As a practical note, it is common for text files to have a large number of blank lines, resulting in a high value of R. The related Unix utility ***bdiff*** differences files too large to fit in RAM by splitting them into smaller segments, differencing the segments, and combining the results; however, this strategy does not produce a min-cost edit script in general. Myers's $O(ND)$ algorithm has been modified to yield an $O(NP)$ algorithm, where P is the number of deletions in an optimal edit script [25]. This algorithm uses the path-compression technique used for shortest-path problems [17, 8] and is especially efficient when one of the inputs is much smaller than the other.

For tree comparison, several edit models have been proposed. For ordered trees with insertions, deletions, and updates, and a cost model that satisfies a triangle inequality, Zhang and Shasha's algorithm is $O(MNb)$, where b is the product of $\min(\text{depth}(T_i), \text{leaves}(T_i))$ for the input trees T_1 and T_2 [26]. The complexity can be lowered for a simplified cost model [16]. These algorithms use definitions of node insertion and deletion that are more general than ours. For example, they permit a node to be deleted without deleting its descendants. The differencing problem for unordered trees is known to be NP-hard, with efficient algorithms for restricted cases [27].

All the above algorithms are for main memory. To our knowledge, there has been very little work on external memory differencing algorithms. In earlier work, we have presented an $O(MN)$ external-memory tree-differencing algorithm [3]. Unlike the methods in this paper, that algorithm requires a $O(MN)$ I/Os even when the two inputs are identical, while the Algorithm rdiff of this paper performs only $M + N$ I/Os.

The problems of differencing and pattern matching are closely related, and it may be possible to use ideas from tree matching methods (e.g, [22, 20, 21]) to design tree differencing algorithms.

6 Conclusion

We studied two formulations of the differencing problem. The first applies to sequence data that are edited using insert and delete operations of unit cost. The second applies to data modeled as rooted, ordered, labeled trees that are edited using node insertions, deletions, and updates, with arbitrary costs (with lower bound 1). We generalized to external memory the diagonal-based computation method that forms the basis of a class of efficient main-memory differencing algorithms. We studied the performance of our external-memory sequence comparison algorithm and discussed extensions for streaming and tree-structured data. We presented a very efficient algorithm for differencing streaming hierarchical data that performs no I/Os other than those required to read its inputs. We analyzed the performance of this algorithm and also studied a variant that works with very little RAM.

As continuing work, we are studying further applications of the diagonal-based differencing technique to external memory and streaming algorithms. We also plan to devise cache-conscious main-memory differencing algorithms. Differencing algorithms form the foundation of a larger change management system that includes differencing operators as primitives. We are also working on incorporating a difference operator into general-purpose query language and on devising an algebra for such use.

References

[1] A. Aho, D. Hirschberg, and J. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the Association for Computing Machinery*, 23(1):1–12, Jan. 1976.

[2] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, June 1997.

[3] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 90–101, Edinburgh, Scotland, Sept. 1999.

[4] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 4–13, Orlando, Florida, Feb. 1998.

[5] Y. Chen, F. K. H. A. Dehne, T. Eavis, and A. Rau-Chaplin:. Parallel ROLAP data cube construction on shared-nothing multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, June 1990.

[7] B. Eaglestone, B. C. Desai, R. Holton, and E. Gulati. Temporal database support for cooperative creative work. In *Proceedings of the Database Engineering and Applications Symposium (IDEAS)*, pages 266–275, Cardiff, Wales, 1998.

[8] F. Hadlock. Minimum detour methods for string or sequence comparison. *Congr. Numer.*, 61:263–274, 1988.

[9] J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

[10] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proceedings of the International Conference on Very Large Data Bases*, Bombay, India, September 1996.

[11] W. Masek and M. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[12] W. Miller and E. Myers. A file comparison program. *Software–Practice and Experience*, 15(11):1025–1040, 1985.

[13] E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[14] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[15] S. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, Dec. 1977.

[16] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11:581–621, 1990.

[17] E. Simon and P. Valduriez. Integrity control in distributed database systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 621–632, 1986.

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

[19] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of the Association of Computing Machinery*, 21(1):168–173, January 1974.

[20] J. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: some preliminary results. In *Proceedings of the ACM SIGMOD Conference*, pages 115–125, May 1994.

[21] J. Wang, D. Shasha, G. Chang, L. Relihan, K. Zhang, and G. Patel. Structural matching and discovery in document databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 560–563, 1997.

[22] J. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, Aug. 1994.

[23] C. Wong and A. Chandra. Bounds for the string editing problem. *Journal of the Association for Computing Machinery*, 23(1):13–16, Jan. 1976.

[24] C. H. Wu, H. Huang, L. Arminski, et al. The Protein Information Resource: an integrated public resource of functional annotation of proteins, 2002. Nucleic Acids Ressearch 30,35-37.

[25] S. Wu, U. Manber, and G.Myers. An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35:317–323, September 1990.

[26] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

[27] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 1995.