Comparing Hierarchical Data in External Memory

Sudarshan S. Chawathe
Department of Computer Science
University of Maryland
College Park, MD 20904
chaw@cs.umd.edu

Abstract

We present an external-memory algorithm for computing a minimum-cost edit script between two rooted, ordered, labeled trees. The I/O, RAM, and CPU costs of our algorithm are, respectively, $4mn+7m+5n$, $6S$, and $O(MN+(M+N)S^{1.5})$, where M and N are the input tree sizes, S is the block size, $m = M/S$, and $n = N/S$. This algorithm can make effective use of surplus RAM capacity to quadratically reduce I/O cost. We extend to trees the commonly used mapping from sequence comparison problems to shortest-path problems in edit graphs.

1 Introduction

We study the problem of comparing snapshots of data to detect similarities and differences between them. Such *differencing* of data has **applications** in version control, incremental view maintenance, data warehousing, standing queries (subscriptions), and change management [Tic85, LGM96, CAW98]. The RCS version control system [Tic85] uses the *diff* program [MM85] to compute and store only the differences between the new and old versions of data that is checked in. As another version control application, consider the process used to merge two divergent versions of a program or document (e.g., the ediff/emerge function in Emacs). The first step consists of comparing the files containing the two versions to determine where and how they differ. These differences are then presented using a graphical interface that allows a user to determine which variant to keep in the merged file.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 25th VLDB Conference,
Edinburgh, Scotland, 1999.**

Differencing algorithms also play a key role in change management systems such as C^3 [CAW98]. Since many databases, especially those on the Web, do not offer change notification facilities, changes must be detected by comparing old and new results of a query. Once changes have been detected in this manner, C^3 uses them to implement standing queries based on the current state as well as the history of the databases being monitored.

We can also use differencing algorithms to reduce the amount of data transmitted over a network in mirroring applications. Popular Web and FTP servers often have dozens of mirror sites around the world. Changes made to the master server need to be propagated to the mirror sites. Ideally, the persons or programs making changes would keep a record of exactly what data was updated. However, in practice, due to the autonomous and loosely organized nature of such sites, there is no reliable record of changes. Further, even if such a record is available, it may be based on a version that is different from the version currently at a certain mirror site. Due to such difficulties, efficient mirroring requires differencing algorithms that compute and propagate only the difference between the version at the master server and that at a mirror site. Similar ideas enable differencing algorithms to improve efficiency in a data warehousing environment [LGM96].

Differencing algorithms are also used to find, mark-up, and browse changes between two or more versions of a document [CRGMW96, CGM97]. Suppose we receive an updated version of an online manual. Again, in the ideal case the new version would highlight the way it differs from the old one. However, for reasons similar to those stated above, in practice we often need to detect the differences ourselves by comparing the two versions. For example, [CAW98] describes experiences in detecting and browsing differences between different versions of a restaurant review database on the Web, while [Yan91] describes the implementation of an application that highlights differences between program versions.

There is a substantial body of prior work on differencing algorithms. The main **distinguishing features** of the work in this paper are the following. (See Section 6 for a detailed discussion.)

- We study algorithms for computing differences between snapshots of **hierarchically structured** data, modeled using rooted, ordered, labeled trees. Our model allows us to accurately capture the hierarchical structure inherent in data such as source code, object class hierarchies, structured documents, HTML, XML, and SGML. For example, an online manual typically has a well-defined hierarchical structure consisting of chapters, sections, subsections, paragraphs, and sentences. Algorithms that take this structure into account produce results that are more meaningful than those that treat their inputs as flat strings.

  While the problem of differencing strings and sequences has been thoroughly studied and admits several efficient solutions, the problem of differencing trees remains challenging. Several formulations of this problem are NP-hard [ZWS95]. In this paper, we study a simple variation that admits efficient solutions.

- We do not assume that the snapshots being differenced are small enough to fit entirely in main memory (RAM); instead, they reside in **external memory** (disk). For example, online manuals for complex machinery, aircrafts, and submarines are tens or hundreds of gigabytes in size, making it impracticable to use main-memory differencing algorithms to compare their versions.

  When data resides in external memory, the number of input-output operations (I/Os), and not the number of CPU cycles, is the primary determinant of running time. Therefore, **external-memory algorithms** use techniques that try to minimize the number of I/Os. A secondary but important consideration is the amount of buffer space required in RAM. See [Vit98] for an overview of external memory algorithms. In this paper, we analyze algorithms based on their I/O, RAM, and CPU costs.

As an illustration of the importance of using an algorithm that is cognizant of the hierarchical structure of data, consider the following example from [Yan91]. Figure 1 depicts depicts fragments of two program versions that are being compared. A sequence comparison program such as the one in[MM85] compares the inputs line-by-line and may result in matching program text as suggested by the solid lines in the figure. Given the nested structure of the program fragment, it is clearly more meaningful to match the inputs as suggested by the dashed lines in the figure. However, the definition of optimality used by most sequence comparison algorithms (based on a longest common subsequence) considers the solution depicted using solid lines more desirable [Mye86]. By modeling the hierarchical structure of programs, tree differencing algorithms are able to produce more meaningful results.

We now present a brief, informal definition of the differencing problem we study in this paper. (See Section 2 for details.) A *rooted, ordered, labeled tree* is a tree in which each node has a label and in which the order amongst

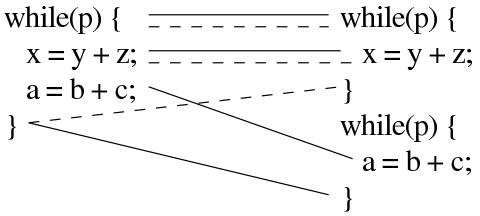

Figure 1: Importance of hierarchical structure

siblings is significant. (The label of a node intuitively represents the data content at that node; it is not a unique key or object identifier.) Trees can be transformed using three *edit operations*: (1) We can *insert* a new leaf node at a specified location in the tree. (2) We can *delete* an existing leaf node. (3) We can *update* the label of a node. Note that the restriction that (1) and (2) operate only on leaf nodes means that to delete an interior node, we must first delete all its descendants; similarly, we must insert a node before inserting any of its descendants.

An *edit script* is a sequence of edit operations that are applied in the order listed. We associate a cost with each edit operation and define the *cost of an edit script* to be the sum of the costs of its component operations.

Problem Statement (informal): Given two rooted, ordered, labeled trees A and B, find a minimum-cost edit script that transforms A to B.

Given trees A and B, we can transform one to the other using any of an infinite number of edit scripts. (For example, given any edit script that transforms A to B, we can append to it operations that insert and immediately delete a node, thus generating an infinite number of edit scripts.) This fact motivates the minimum-cost requirement in the problem definition. Another motivation for the minimum-cost requirement is the following: Given two trees that differ only in one node label, the intuitively desirable edit script is one that contains a single update operation. We need to weed out edit scripts that unnecessarily insert, delete, and update nodes. An edit script of minimum cost cannot contain such redundant or wasteful edit operations (since we can obtain another edit script with lower cost by getting rid of the redundancies and inefficiencies).

The two main **contributions** of this paper may be summarized as follows:

- We present an efficient external-memory algorithm for computing the difference (minimum-cost edit script) between two snapshots of hierarchical data (trees). The I/O, RAM, and CPU costs of our algorithm are, respectively, $4mn + 7m + 5n$, $6S$, and $O(MN\sqrt{S})$, where M and N are the input tree sizes, S is the block size, $m = M/S$, and $n = N/S$. To our knowledge, this algorithm is the first external-memory differencing algorithm (for sequences or trees). The $O(mn)$ I/O complexity of our algorithm is optimal over a wide class of computation models due to the $O(mn)$ lower bound for the sequence comparison problem (which

is a simple special case of our tree comparison problem) [AHU76, WC76].

- We reduce our tree comparison problem to a shortest path problem in a well-studied graph called the edit graph. This reduction opens the door for generalizing to trees several efficient sequence comparison algorithms that are based on edit graphs (e.g., [MM85, Mye86, WMG90]).

Outline of the paper: In Section 2, we describe our model of trees, edit operations, and edit scripts, followed by the formal problem statement. Section 3 briefly describes edit graphs as they are used for sequence comparison and then presents our modifications that allow them to be used for tree comparison. In Section 4, we present a simple main-memory algorithm for our tree comparison problem. This algorithm illustrates the use of our edit graphs and serves as a basis for our external-memory algorithm. Section 5 first explores a naive extension of our main-memory algorithm for external memory and then presents our main algorithm, *xmdiff*. Related work is discussed in Section 6, followed by the conclusion in Section 7.

2 Model and Problem Statement

Hierarchical data is naturally modeled by trees. In this paper, we focus on *rooted, ordered, labeled trees*. Each node in such a tree has a label associated with it. Informally, we can think of the label of a node as its data value. The children of a node are totally ordered; thus, if a node has k children, we can uniquely identify the ith child, for $i = 1 \ldots k$. There is a distinguished node called the root of the tree. (This feature distinguishes these trees from acyclic graphs, which are also called free trees.)

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 $T_1, \ldots, T_k$, 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 $l(n)$ [Sel77]. 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**.

In the rest of this paper, we use the term trees to mean rooted, ordered, labeled trees. Figure 2 depicts several such trees. The letter next to a node suggests its label. The number next to a node is a node identifier. Note that when we are given two trees to compare, there is, in general, no correspondence between the node identifiers. (In fact, computing such a correspondence is equivalent to computing a minimum-cost edit script for our formulation of the problem.) For example, when we are comparing two versions of a manual, the node identifiers in Figure 2 may represent offsets within the SGML source files for the manuals. Since the source files for the manuals are separate, there is no correspondence between these offsets.

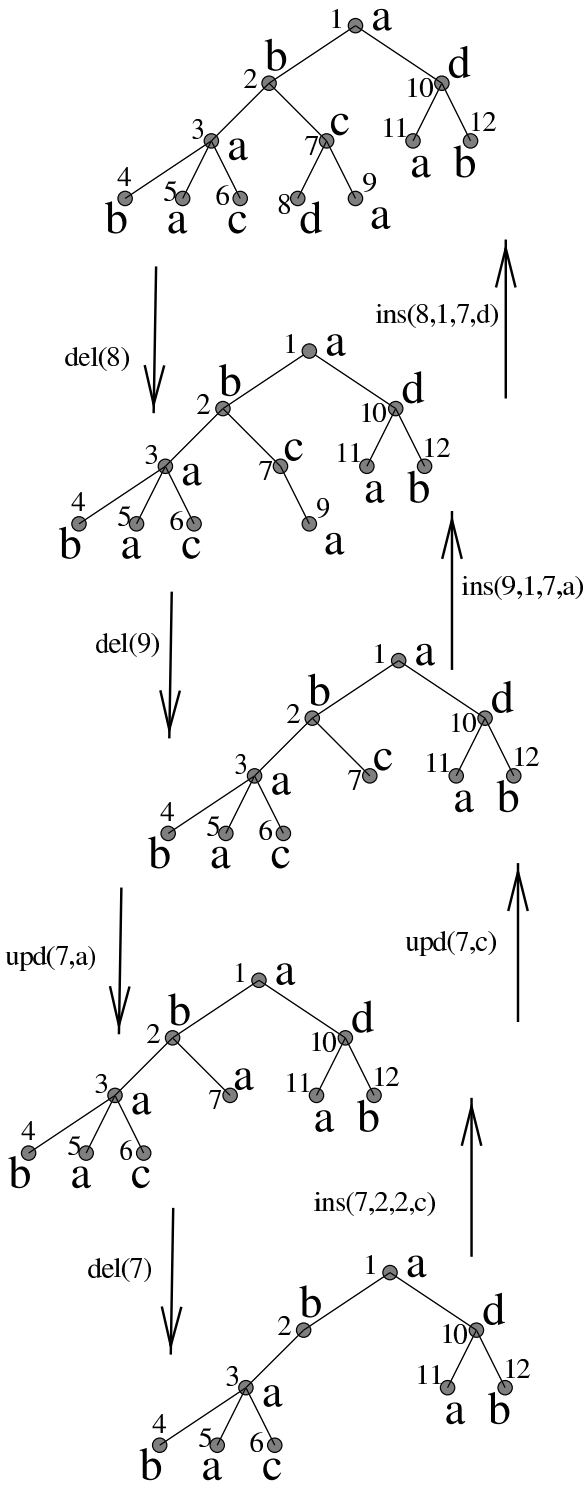

Figure 2: Edit operations on trees

We model changes to trees using the following **tree edit operations**:

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 $i \in [1 \ldots k+1]$. The insertion operation *ins*(n, i, p, l) inserts the node n as the ith 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 ith 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 cannot be deleted or inserted.

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. Consider Figure 2. The downward arrows illustrate the application of the following edit script: *del*(8), *del*(9), *upd*$(7, a)$, *del*(7). Similarly, the upward arrows illustrate the application of another edit script.

As discussed in Section 1, we formalize the desirability of edit scripts that perform as few and as small changes as possible by defining a cost model for edit scripts. Let $c_i(x)$ and $c_d(x)$ be arbitrary functions return a positive number representing the costs of, respectively, inserting and deleting a node x. Similarly, the cost of updating a label l_1 to l_2 is given by $c_u(l_1, l_2)$. The cost of an edit script is the sum of the costs of its component operations. We can now formally define the problem of **differencing trees** as follows:

Problem Statement: 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.

3 Edit Graphs

In this section, we introduce an auxiliary structure, called an edit graph, that we later use in our differencing algorithms. Edit graphs have been used by several efficient algorithms for comparing sequences (equivalently, strings) [Mye86, MM85, WMG90]. In effect, the problem of finding a minimum-cost edit script between two sequences is reduced to the problem of finding a shortest path from one end of the edit graph to the other. Since an edit graph has a very simple and regular structure, this shortest path problem can typically be solved very efficiently. Below, we first explain how edit graphs are used for sequence comparison and then introduce our modifications that permit them to be used for comparing trees.

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 3. (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 3 corresponds to the sequences (strings) $A = ababaccdadab$ and $B = acabbdbbabc$.

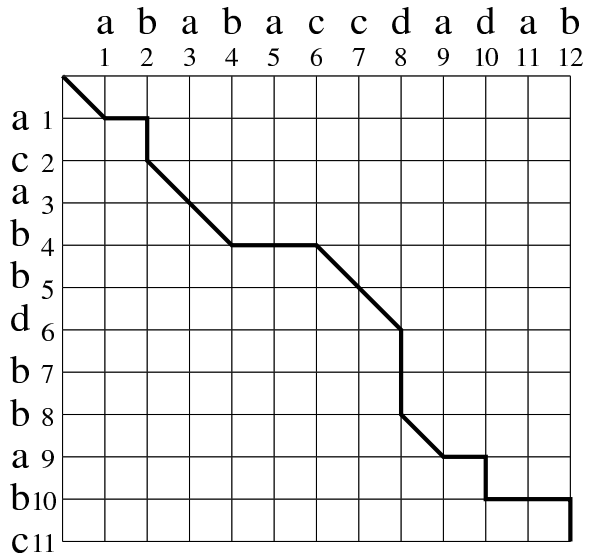

Figure 3: Edit graph for sequence comparison

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 $((x-1, y-1), (x, y))$ 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 $c_u(a_x, b_y)$.

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. For details, see [Mye86].

For the example suggested in Figure 3, the highlighted path corresponds to the following edit script:

del$(A[2])$, *ins*$(B[2])$, *del*$(A[5])$, *del*$(A[6])$,

$upd(A[7], b), del(A[7]), del(A[8]), del(A[10]),$
$ins(B[10]), del(A[11]), del(A[12]), ins(B[11])$

It is easy to verify that applying the above script to A produces B.

In order to use edit graphs to compare trees, we need to modify their definition to incorporate the constraints imposed by the structure of the trees being compared. For example, we must model the constraint that if an interior node is deleted then all its descendants must also be deleted. Before we proceed, we need to define the *ld-pair* representation of a tree.

We define the **ld-pair** of a tree node to be the pair (l, d), where l is the node's label and d is its depth in the tree. We use $p.l$ and $p.d$ to refer to, respectively, the label and depth of an ld-pair p. The **ld-pair representation of a tree** is the list, in preorder, of the ld-pairs of its nodes. In the rest of this paper, we assume that trees are in the ld-pair representation. (A tree can be converted to this representation using a single preorder traversal.)

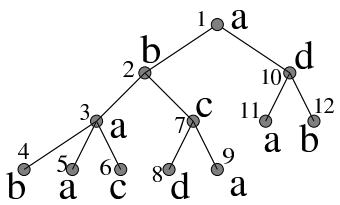

Figure 4: Input tree A

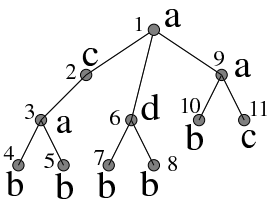

Figure 5: Input tree B

Consider the trees A and B depicted in Figures 4 and 5. The letter next to each node suggests its label and the number next to a node is its preorder rank, which also serves as its identifier. The ld-pair representations of A and B are as follows:

$$A = ((a,0),(b,1),(a,2),(b,3),(a,3),(c,3),(c,2),(d,3),(a,3),(d,1),(a,2),(b,2))$$
$$B = ((a,0),(c,1),(a,2),(b,3),(b,3),(d,1),(b,2),(b,2),(a,1),(b,2),(c,2))$$

Given a tree (in ld-pair representation) $A = (a_1\ a_2\ \ldots\ a_M)$, we use the notation $A[i]$ to refer to the ith node a_i of tree A. Thus, $A[i].l$ and $A[i].d$ denote, respectively, the label and the depth of the ith node of A.

We define **the edit graph of two trees** A and B to consist of a $(M+1) \times (N+1)$ grid of nodes as suggested by Figure 6. There is a node at each (x, y) location for $x \in [0 \ldots (M+1)]$ and $y \in [0 \ldots (N+1)]$. These nodes are connected by directed edges as follows:

- For $x \in [0, M-1]$ and $y \in [0, N-1]$, there is a diagonal edge $((x, y), (x+1, y+1))$ if and only if $A[x+1].d = B[y+1].d$. (For clarity, these edges are omitted in Figure 6.)
- For $x \in [0, M-1]$ and $y \in [0, N]$, there is a horizontal edge $((x, y), (x+1, y))$ unless $y < N$ and $B[y+1].d > A[x+1].d$.
- For $x \in [0, M]$ and $y \in [0, N-1]$, there is a vertical edge $((x, y), (x, y+1))$ unless $x < M$ and $A[x+1].d > B[y+1].d$.

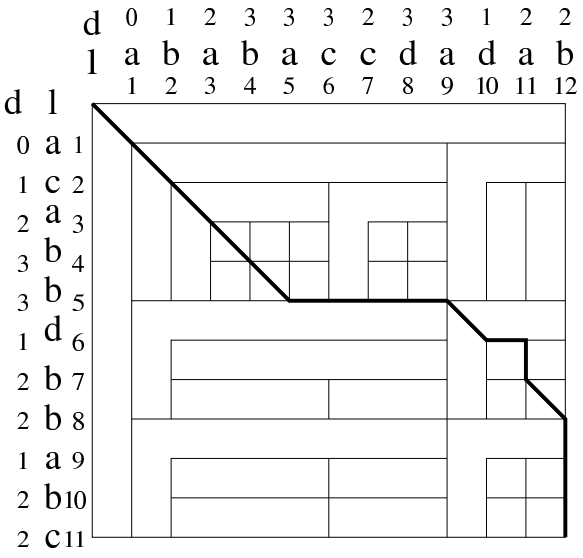

Figure 6: Edit graph for tree comparison

As was the case for sequence edit graphs, horizontal edges represent deletions, vertical edges represent insertions, and diagonal edges represent matching of nodes (with an update operation required if the labels of the matched nodes differ). Similarly, each edge has an edge weight equal to the cost of the corresponding edit operation. However, in contrast to sequence edit graphs, several horizontal and vertical edges are missing from tree edit graphs. Intuitively, the missing vertical edges ensure that once a path in the edit graph traverses an edge signifying the deletion of a node n, that path can only continue by traversing edges that signify the deletion of all the nodes in n's subtree. Similarly, the missing horizontal edges ensure that any path that traverses an edge signifying the insertion of a node n can only continue by traversing the edges signifying insertion of all nodes in n's subtree.

We can 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 tree edit graph; conversely, every path from $(0, 0)$ to (M, N) corresponds to an edit script that transforms A to B.

For example, the path indicated using bold lines in Figure 6 corresponds to the following edit script, where $n_1, \ldots, n_4$ are arbitrary identifiers for the newly inserted nodes:

$$upd(2, c), upd(5, b), del(6), del(7), del(8), del(9)$$
$$del(11), ins(n_1, 1, 10, b), ins(n_2, 3, 1, a)$$
$$ins(n_3, 1, n_2, b), ins(n_4, 2, n_2, c)$$

4 Differencing in Main Memory

In Section 3, we reduced the problem of computing a min-cost edit script between two trees to the problem of finding a shortest path in the edit graph of those trees. We now use that reduction to present a main-memory algorithm for differencing trees. Given a $(M+1) \times (N+1)$ edit graph G, let D be a $(M+1) \times (N+1)$ matrix such that $D[x, y]$ is the length of a shortest path from $(0, 0)$ to (x, y) in the edit graph. We call D the **distance matrix** for G.

For notational convenience, let us define the weight of an edge that is missing from the edit graph to be infinity. Consider any path that connects the origin $(0, 0)$ to $n = (x, y)$ in the edit graph. Given the graph's structure, the previous node on this path is either the node to the left of n, the node above n or the node diagonally to the left and above n. Therefore, the distance of n from the origin cannot be greater than that distance for the node to its left plus the weight of the edge connecting the left neighbor to n. Similar relations hold for the top and diagonal neighbors of n, yielding the following recurrence for $D[x, y]$, where $0 < x \leq M$ and $0 < y \leq N$:

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

This recurrence leads to the following dynamic-programming algorithm for computing the distance matrix D. We call this algorithm mmdiff, for main-memory differencing.

Algorithm mmdiff

Input: Arrays A and B, which represent two trees in ld-pair representation. Thus $A[i].l$ and $A[i].d$ denote, respectively, the label and depth of the ith node (in preorder) of A (and analogously for B). The number of elements in A and B is M and N, respectively.
Output: The distance matrix D where $D[i, j]$ equals the length of the shortest path from $(0, 0)$ to (i, j) in the edit graph of A and B.
Method: Figure 7 presents the pseudocode for a Algorithm mmdiff. All the pseudocode in this paper assumes short-circuit evaluation of Boolean expressions. We initialize D at the origin $(0, 0)$, followed by computation of distances along the top and left edge of the matrix. The nested for loop is a direct implementation of the recurrence for $D[i, j]$. If we assume that the functions c_u, c_d, and c_i execute in constant times α, β, and γ, respectively, the running time of mmdiff is proportional to $1+\alpha M+\beta N+(\alpha+\beta+\gamma)MN$, or $O(MN)$.

```
D[0,0] := 0;
for i := 1 to M do
    D[i,0] := D[i-1,0] + c_d(A[i]);
for j := 1 to N do
    D[0,j] := D[0,j-1] + c_i(B[j]);
for i := 1 to M do
    for j := 1 to N do begin
        m1 := ∞; m2 := ∞; m3 := ∞;
        if (A[i].d = B[j].d) then
            m1 := D[i-1,j-1] + c_u(A[i], B[j]);
        if (j = N or B[j+1].d ≤ A[i].d) then
            m2 := D[i-1,j] + c_d(A[i]);
        if (i = M or A[i+1].d ≤ B[j].d) then
            m3 := D[i,j-1] + c_i(B[j]);
        D[i,j] := min(m1, m2, m3);
    end;
```

Figure 7: Algorithm mmdiff

Once we have computed the distance matrix D, it is easy to recover from it a shortest path from $(0, 0)$ to (M, N). We start at (M, N) and trace the recurrence relation for D backwards. As we traverse horizontal, vertical, and diagonal edges, we emit the appropriate deletion, insertion, and update operation, respectively. Thus, we have the following algorithm for recovering an edit script from the distance matrix. We call this algorithm mmdiff-r (for mmdiff-recovery).

Algorithm mmdiff-r

Input: Arrays A and B representing the input trees (as in Algorithm mmdiff) and the distance matrix D computed by mmdiff.
Output: A min-cost edit script that transforms A to B. (For simplicity of presentation, the insertion and deletion operations only identify the nodes being inserted and deleted. Using the information from the original trees, it is easy to generate an edit script in the syntax of Section 2. Further, insertions of interior nodes are printed after the insertions of their descendants; this ordering is easily fixed.)
Method: Figure 8 presents the pseudocode for Algorithm mmdiff-r. At each iteration of the while loop i and/or j is decremented by one. Thus there are between $\max(M, N)$ and $M + N$ iterations of the while loop, with each iteration performing a constant amount of work. Thus, the running time of mmdiff-r is $O(M + N)$.

```
i := M; j := N;
while (i > 0 and j > 0) do
    if (D[i, j] = D[i − 1, j] + c_d(A[i]) and
            (j = N or B[j + 1].d ≤ A[i].d)) then begin
        print("del" i);
        i := i − 1;
    end;
    else if (D[i, j] = D[i, j − 1] + c_i(B[j]) and
            (i = M or A[i + 1].d ≤ B[j].d)) then begin
        print("ins" j);
        j := j − 1;
    end;
    else begin
        if(A[i].l ≠ B[j].l) then print("upd" i B[j].l);
        i := i − 1; j := j − 1;
    end;
while (i > 0) do begin
    print("del" i);
    i := i − 1;
end;
while (j > 0) do begin
    print("ins" j);
    j := j − 1;
end;
```

Figure 8: Algorithm mmdiff-r

5 Differencing in External Memory

In Section 4, we presented algorithms mmdiff and mmdiff-r to compute a minimum-cost edit distance between two trees in main memory. These algorithms require RAM space for not only the input trees A and B, but also for the distance matrix D. If A and B have sizes M and N, respectively, then the distance matrix D is of size MN, which can be prohibitively large for even modestly sized inputs that fit in RAM. In situations where A and B themselves are too large to fit in RAM, the problem is much worse.

Let us first consider a naive extension of Algorithm mmdiff for external memory. We use two buffers, B_1 and B_2, to read, as needed, the trees A and B, respectively, from disk into RAM. The buffers are exactly large enough to hold one disk block (of size S). As the distance matrix D is computed, we write the distances to a third buffer B_3, also of size S. When B_3 is completely filled, we write it out to disk and overwrite it in RAM. Using a similar buffering scheme, we can adapt the algorithm mmdiff-r, used for recovering the edit script, for external memory. Let us call these buffered version of mmdiff and mmdiff-r **Algorithm bmdiff** and **Algorithm bmdiff-r**, respectively.

Since Algorithm bmdiff computes the distance matrix D in column-major order, each of the $\lceil N/S \rceil = n$ blocks of B is read once for each of the M nodes in A, incurring an I/O cost of Mn. On the other hand, each of the $\lceil M/S \rceil = m$ blocks of A is read exactly once, for an I/O cost of m. Algorithm bmdiff also stores the entire distance matrix, of size MN to disk, incurring an I/O cost of MN/S. Thus the total I/O cost of algorithm bmdiff is $Mn+m+MN/S$, or approximately $2Smn+m$. In addition to space for temporary variables Algorithm bmdiff needs space for only the three buffers B_1, B_2, and B_3, of size S each. Thus the RAM cost of Algorithm bmdiff is $3S$. Other than operations required to read and write disk blocks, Algorithm bmdiff performs the same operations as Algorithm mmdiff. Thus its CPU cost is $O(MN)$. Thus, we may summarize **Algorithm bmdiff's performance** as follows:

I/O	RAM	CPU
$2Smn + m$	$3S$	$O(MN)$

By using techniques similar to those used for computing nested-loop joins in relational databases, it is easy to reduce the I/O cost of reading A and B from Mn to mn. Intuitively, instead of computing the distance matrix in column-major order, we compute it in a blocked manner: When we read in the Ith block of A and the Jth block of B, we compute the entire $S \times S$ submatrix $\{D[i, j] : i \in [SI, (I+1)-1], j \in [SJ, S(J+1)-1]\}$.

However, it is not obvious how we can improve on the I/O cost of storing the distance submatrix itself, which is of size MN/S blocks. As described below, our approach is to avoid storing all of the distance matrix, storing instead only a coarse grid from which the rest of the distance matrix can be quickly computed.

5.1 Computing the Distance Matrix

Consider the $(M+1)\times(N+1)$ distance matrix D suggested by the arrangement of dots in Figure 9, corrsponding to input trees A and B of sizes M and N, respectively. As suggested in the figure, we divide D into a set of overlapping **tiles.** For clarity, the figure alternates the use of solid and dashed lines for marking the tiles. The overlap between neighboring tiles is one item wide.

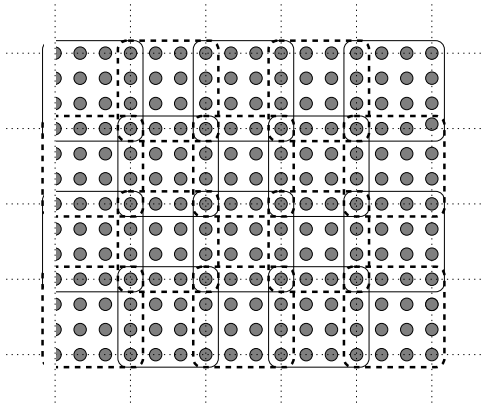

Figure 9: Tiling the distance matrix

More precisely, let $S' = S - 1$. For simplicity in presentation, we shall henceforth assume that M and N are both integral multiples of S', with $M = mS'$ and $N = nS'$. The distance matrix D is then divided into

mn tiles laid out in m columns and n rows. We number these rows and columns of tiles starting with 0, and use the notation (i_b, j_b) to denote the tile in the i_bth column (of tiles) and j_bth row(of tiles). Thus, the tile (i_b, j_b) consists of the following submatrix of the distance matrix D: $\{D[x,y] : x \in [Si_b..S(i_b+1)-1], y \in [Sj_b..S(j_b+1)-1]$.

Our external memory algorithm, called xmdiff (for external memory differencing), is based on the following three key ideas: First, instead of storing the entire distance matrix D, we store only the top and left edges of each of the tiles described above. We call this grid consisting of the tile edges the **distance grid.** Each tile edge is of size S, and there are mn tiles in all. Thus the entire distance grid can be stored using only $2Smn/S = 2mn$ blocks of storage. Referring back to Figure 9, only those entries in the the matrix that are crossed by a dotted line are stored on disk. (Note that for clarity, Figure 9 assumes that the dimension of each tile is only 4×4 items, corresponding to $S = 4$. For such low values of S, our technique does not appear to yield much savings. However, for typical values of S that lie in the hundreds, the savings are substantial.)

Second, 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 for D in Section 3. Further, using a few temporary variables, this recurrence allows us to update in-place an array that initially contains the distances for the top (left) edge of a tile to one containing the distances for the bottom (respectively, right) edge.

Third, given the distance grid stored on disk, a shortest path in the edit graph from $(0,0)$ to (M,N) (and thus the corresponding min-cost edit script that transforms tree A to tree B) can be computed in linear time by working backwards from (M,N), as done in Algorithm mmdiff-r. A straightforward algorithm based on this idea requires $O(S^2)$ RAM as working store. However, we also describe an enhanced version of the algorithm that requires only $O(S)$ RAM.

We assume that the first block of 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. These assumptions are not necessary for our algorithm; we make them only to simplify the following presentation of the algorithm.

Recall, from Section 3, that we represent a tree using the preorder listing of the ld-pairs of its nodes. These ld-pair sequences are packed densely into disk blocks when the trees are represented on disk. Nodes within a block are arranged at fixed offsets beginning with 0.

Disk blocks are read and written using the function RdBlk and the procedure WrBlk, respectively. RdBlk takes as arguments a file name (which we use to group related blocks) and a block number. Block numbers for each file are separate, and are numbered sequentially beginning with 0. For example, the ith block of a file called A is denoted by $A(i)$ and is read using RdBlk(A,i). The notation for WrBlk is analogous to that used for RdBlk: WrBlk(A,i) writes the buffer A to the ith block of file A. In addition to one dimensional indexing of disk blocks, we also use two-dimensional indexing of the form $D(i,j)$ where the ranges of i and j are fixed and known in advance. Such double indexing of blocks is only a notational convenience. Using a standard row-major encoding of two-dimensional matrices, $D(i,j)$ is equivalent to $D(iN+j)$ where N is the number of different j values. Thus each RdBlk and WrBlk operation requires only a single I/O.

Algorithm xmdiff

Input: Arrays A and B representing the input trees, containing M and N elements, respectively (as in Algorithm mmdiff).
Output: The distance grid (defined above), giving the length of the shortest path from $(0,0)$ to (x,y) in the edit graph of A and B for all edit graph nodes (x,y) that lie on the edges of tiles of dimension $S \times S$.
Method: Figure 10 lists the pseudocode for Algorithm xmdiff. The first part of the algorithm consists of two for loops which correspond to the first two for loops of Algorithm mmdiff in Figure 7. The first for loop in Figure 10 computes and writes to disk the top row of the distance matrix. Similarly, the second for loop computes and writes to disk the leftmost column of the distance matrix.

The second part of Algorithm xmdiff (the third for loop in Figure 10) corresponds to the nested for loops of Algorithm mmdiff in Figure 7. Each iteration of the inner loop computes the distances in one tile (submatrix) of the distance matrix. Recall that we identify a tile by its tile column and tile row numbers, with numbering starting at 0. The innermost loop computes distances in tile (i_b, j_b). This computation uses, in addition to the appropriate blocks of the input trees A and B, the distances for nodes on the top and left edges of the tile. Due to the overlap of tiles, these edges are the bottom and right edges of some other (previously computed) tiles. The distances for nodes on the bottom and right edges of the tile are then written to disk. By carefully ordering the distance calculations and by using temporary variables (t_a and t_b), we are able to update in place the array D_a, which initially contains the distances for nodes on the top edge, to the distances for nodes in the bottom edge. Similarly, the array D_b initially containing the distances of nodes in the left edge is updated in place to the distances of nodes in the right edge. The tests in the if statements are analogous to those in Algorithm mmdiff.

Analysis

Algorithm xmdiff uses only four buffers in RAM: A, B, D_a, and D_b, each of size S. Thus, the RAM storage requirements are only $4S$ (in addition to the small, constant amount of storage needed for program code and scalar variables).

Let $S' = S - 1$, $m = M/S'$, and $n = N/S'$. The first two for loops of the algorithm make a total of $m + n$ calls to the each of the procedures RdBlk and WrBlk, giving $2(m+n)$ as the I/O cost. In the nested for loops, the first

```
S' := S − 1;
for i_b := 0 to (M/S' − 1) do begin
    A := RdBlk(A, i_b);
    if(i_b > 0) then D_a[0] := D_a[S'];
    else D_a[0] := 0;
    for i := 1 to S' do
        D_a[i] := D_a[i − 1] + c_d(A[i]);
    WrBlk(D_a, i_b, 0);
end;
for j_b := 0 to (N/S' − 1) do begin
    B := RdBlk(B, j_b);
    if(j_b > 0) then D_b[0] := D_b[S'];
    else D_b[0] := 0;
    for j := 1 to S' do
        D_b[j] := D_b[j − 1] + c_i(B[j]);
    WrBlk(D_b, 0, j_b);
end;
for i_b := 0 to (M/S' − 1) do begin
    A := RdBlk(A, i_b);
    D_a := RdBlk(D_a, i_b, 0);
    for j_b := 0 to (N/S' − 1) do begin
        B := RdBlk(B, j_b);
        D_b := RdBlk(D_b, i_b, j_b);
        D_a[0] := D_b[S'];
        for i := 1 to S' do begin
            t_b := D_b[0];
            D_b[0] := D_a[i];
            for j := 1 to S' do begin
                m_1 := ∞; m_2 := ∞; m_3 := ∞;
                if (A[i].d = B[j].d) then
                    m_1 := t_b + c_u(A[i], B[j]);
                if (j = S' or B[j + 1].d ≤ A[i].d) then
                    m_2 := D_b[j] + c_d(A[i]);
                if (i = S' or A[i + 1].d ≤ B[j].d) then
                    m_3 := D_b[j − 1] + c_i(B[j]);
                t_b := D_b[j];
                D_b[j] := min(m_1, m_2, m_3);
            end;
            D_a[i] := D_b[S'];
        end;
        WrBlk(D_a, i_b + 1, j_b);
        WrBlk(D_b, i_b, j_b + 1);
    end;
end;
```

Figure 10: Algorithm xmdiff

two RdBlk statements (for A and D_a) are executed m times. The next two RdBlk statements (for B and D_b) are executed mn times. Finally, the WrBlk statements are executed mn times. Thus, the total number of I/Os in the nested loops of xmdiff is $2m + 4mn$. Combining this number with that for the first two for loops, we conclude that xmdiff makes $4mn + 4m + 2n$ I/Os.

Finally, it is easy to observe that the CPU time is $O(MN)$. Thus, we can summarize **Algorithm xmdiff's performance** as follows, where M and N are the sizes of the input trees, S is the block size, $m = M/(S-1)$, and $n = N/(S-1)$:

I/O	RAM	CPU
$4mn + 4m + 2n$	$4S$	$O(MN)$

5.2 Recovering the Edit Script

Recall, from Figure 8, the Algorithm mmdiff used to recover a minimum-cost edit script in RAM. Using the distance matrix D as a guide, Algorithm mmdiff-r traverses the shortest path from $(0,0)$ to (M,N) backwards, emitting appropriate edit operations along the way. Unlike mmdiff, algorithm xmdiff does not store the entire distance matrix D, making a direct application of the path-recovery algorithm mmdiff-r impossible. However, for each tile (i,j) of the edit graph, xmdiff stores the distances for nodes on its top and left edges in the disk blocks $D_a(i,j)$ and $D_b(i,j)$, respectively. By reading in these blocks and using algorithm mmdiff, the distance matrix for a tile (i,j) can easily be computed at $O(S^2)$ CPU cost (where S is the block-size). Based on these ideas, we have the following:

Algorithm xmdiff-r

Input: Arrays A and B representing the input trees (as in Algorithm mmdiff) and the distance grid computed by xmdiff, stored on disk as D_a and D_b.
Output: A min-cost edit script that transforms A to B (as in Algorithm mmdiff-r).
Method: Figure 11 presents the pseudocode for xmdiff-r. As in algorithm mmdiff-r, we begin at the node (M,N) of the edit graph and move backwards along the shortest path from $(0,0)$ to (M,N). At each iteration of the outer while loop, the current position (i,j) in the edit graph is moved back to either $(i-1,j)$, $(i,j-1)$, or $(i-1,j-1)$ using the if-then-else statement that is very similar to that in algorithm mmdiff-r. Using the boolean variables n_a and n_b, we detect the situation when the current position (i,j) first moves into a new tile in the horizontal and vertical direction, respectively. When n_a is nonzero, (i,j) has just moved to a new tile to the left of the old tile. In this case, we read in the corresponding block of the input file A. Similarly, when n_b is nonzero, we read in the appropriate block of the input file B. In both cases, we read in the top and left edges of the distance matrix for the new tile, (i_b, j_b), into the arrays T and L, respectively. We use T and L to initialize the top and left edges of the complete distance matrix D for the new tile. Recomputation of the

distances in the rest of the matrix D is done in a manner completely analogous to algorithm mmdiff.

Analysis

Each iteration of the outer while loop decrements at least one of i and j; thus, there are at most $M + N$ iterations of that loop. Since i ranges from M down to 1, it follows that n_a is set to 1 for $M/S' = m$ iterations. Similarly, n_b is set to 1 for $N/S' = n$ iterations. Blocks from D_a and D_b are read in whenever one of n_a and n_b is nonzero. In the worst case, this situation can occur $m+n$ times. Thus the I/O cost due to reading D_a and D_b is at most $2(m + n)$. It is easy to observe that each of the m blocks of the input A is read exactly once, for an I/O cost of m. Similarly, the I/O cost incurred in reading the input B is n. Thus, the total I/O cost of algorithm xmdiff-r is $2(m + n) + m + n = 3(m + n)$.

The significant RAM storage requirements of xmdiff-r are due to the arrays A, B, T, and L, and the distance matrix D. The arrays A, B, T, and L are all of size S, the block size. Unfortunately, the distance matrix D is of size S^2. Thus, the total RAM cost is $4S + S^2$.

It is easy to observe that the CPU cost of algorithm xmdiff-r is $O((M + N)S)$. Thus, **Algorithm xmdiff-r's performance** may be summarized as follows:

I/O	RAM	CPU
$3(m+n)$	$4S+S^2$	$O((M+N)S)$

Reducing the RAM cost of xmdiff-r

We can reduce the S^2 RAM space needed to store the distance matrix D as follows. We divide the $S \times S$ distance matrix D into S subtiles of size $\sqrt{S} \times \sqrt{S}$ as suggested by Figure 12. Consider the process of tracing the shortest path backwards through D. We divide this task into stages corresponding to traversing the subtiles. We begin with the subtile in the lower right corner. At the end of each stage, we move to a subtile that is above and/or to the left of the current subtile. At the beginning of the stage corresponding to a subtile, we compute (as before) the distance matrix D, but store distances for only those points that lie in this subtile. Using a technique similar to that used by algorithm mmdiff, this computation can be performed using only a buffer of size S as working storage. Since the size of a subtile is $\sqrt{S} \times \sqrt{S} = S$, the total storage required for the subtile computation is $2S$. Combined with the $4S$ space required to store A, B, T, and L, we have a total RAM cost of $6S$.

The subtile-based enhancement described above results in the distance matrix for each tile being recomputed several times, thus increasing the CPU cost. After each recomputation, the current subtile moves one position to the left and/or up. Thus, the number of subtile computation stages is between $\sqrt{S}$ and $2\sqrt{S}$ (since there are $\sqrt{S}$ subtiles along each dimension of the distance matrix). It follows that the total CPU cost is therefore increased by a factor of at most $2\sqrt{S}$, to $O((M + N)S^{1.5})$.

```
i := M; j := N; S' := S − 1;
n_a := 1; n_b := 1;
while (i > 0 and j > 0) do begin
    i_b := ⌈i/S'⌉ − 1; j_b := ⌈j/S'⌉ − 1;
    i_i := i mod S'; j_i := j mod S';
    if (n_a > 0) then A := RdBlk(A, i_b);
    if (n_b > 0) then B := RdBlk(B, j_b);
    if (n_a > 0 or n_b > 0) then begin
        T := RdBlk(D_a, i_b, j_b); L := RdBlk(D_b, i_b, j_b);
        n_a := 0; n_b := 0;
        D[0..S', 0] := T; D[0, 0..S'] := L;
        for x := 1 to S' do
            for y := 1 to S' do begin
                m_1 := ∞; m_2 := ∞; m_3 := ∞;
                if (A[x].d = B[y].d) then
                    m_1 := D[x − 1, y − 1] + c_u(A[x], B[y]);
                if (y = S' or B[y + 1].d ≤ A[x].d) then
                    m_2 := D[x − 1, y] + c_i(B[y]);
                if (x = S' or A[x + 1].d ≤ B[y].d) then
                    m_3 := D[x, y − 1] + c_d(A[x]);
                D[i, j] := min(m_1, m_2, m_3);
            end;
    end;
    if (D[i_i, j_i] = D[i_i − 1, j_i] + c_d(A[i_i]) and
            (j_i = S' or B[j_i + 1].d ≤ A[i_i].d)) then begin
        i := i − 1;
        if (i mod S' = 0) then n_a := 1;
        print("del" i);
    end;
    else if (D[i_i, j_i] = D[i_i, j_i − 1] + c_i(B[j_i]) and
            (i_i = S' or A[i_i + 1].d ≤ B[j_i].d)) then begin
        j := j − 1;
        if (j mod S' = 0) then n_b := 1;
        print("ins" j);
    end;
    else begin
        i := i − 1; j := j − 1;
        if (i mod S' = 0) then n_a := 1;
        if (j mod S' = 0) then n_b := 1;
        if(A[i_i].l ≠ B[j_i].l) then print("upd" i B[j].l);
    end;
end;
while (i > 0) do begin
    print("del" i);
    if(i mod S' = 0) then A := RdBlk(A, ⌈i/S'⌉ − 1);
    i := i − 1;
end;
while (j > 0) do begin
    print("ins" j);
    if(j mod S' = 0) then B := RdBlk(B, ⌈j/S'⌉ − 1);
    j := j − 1;
end;
```

Figure 11: Algorithm xmdiff-r

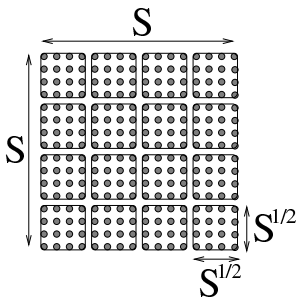

Figure 12: Reducing mmdiff-r's RAM cost

Thus, the **performance of the modified Algorithm xmdiff-r** may be summarized as follows:

I/O	RAM	CPU
$3(m+n)$	$6S$	$O((M+N)S^{1.5})$

Thus, the costs of executing Algorithm xmdiff followed by Algorithm xmdiff-r are as follows:

I/O	RAM	CPU
$4mn+7m+5n$	$6S$	$O(MN+(M+N)S^{1.5})$

Although we have described S to be the block size, our algorithm does not depend on S being equal to the block transfer unit. Another way to interpret the above result is the following: Suppose we have R units of RAM that we can use for our algorithm's buffers. We set $R = 6S$; that is, $S = R/6$. Substituting R for S in our performance results tells us that by increasing the amount of RAM buffer space, we can achieve a quadratic reduction in the significant $4mn$ part of the I/O cost.

6 Related Work

Differencing algorithms have received considerable attention in the research literature, with the problem of comparing sequences receiving the most attention [SK83]. Early sequence comparison algorithms include the classic $O(mn)$ Wagner-Fischer algorithm [WF74], which was shown to be optimal for a wide class of computation models in [AHU76, WC76]. Using the so-called "four Russians" technique, a more efficient $O(nm/\log n)$ algorithm for finite alphabets is presented in [MP80].

More recent work on sequence comparison has focused on improving the expected case running time using output-sensitive algorithms. For example, Myers presents an $O(ND)$ algorithm, where N and D are the sizes of the input and edit script, respectively [Mye86]. (This algorithm forms the basis of the widely used *diff* utility [MM85].) Further improvements are reported in [WMG90]. These algorithms are based on using the special structure of an edit graph.

Our technique for mapping the tree differencing problem to edit graphs, as described in Section 3, allows us to apply the above sequence comparison results to trees. For example, it is relatively straightforward to extend these algorithms in [Mye86] for differencing trees in main memory without any significant increase in running time. However, extending these results to external memory appears more complicated and is part of our continuing work.

Several formulations of the tree comparison problem have also been studied. Early work includes Selkow's recursive algorithm for a simple formulation similar to the one in this paper [Sel77]. More recent work includes [ZS89], which studies ordered trees, and [ZWS95], which studies unordered trees. Most formulations of the tree differencing problem for unordered trees are NP-hard.

There are significant advantages to describing tree differences using not only the three basic edit operations (insert, delete, and update), but also subtree operations such as move, copy, and uncopy. In [CRGMW96], we studied a variation that includes a subtree move operation and described an efficient algorithm that uses simplifying assumptions based on domain characteristics. In [CGM97], we studied a variation that includes subtree copy and uncopy operations in addition to moves. These algorithms are used in the implementation of the C^3 system for managing change in autonomous databases [CAW98].

All the above algorithms are main-memory algorithms; that is, they assume that all input data and working storage resides in RAM. To our knowledge, ours is the first external-memory differencing algorithm. The design of external-memory algorithms based on the problem formulations and ideas in the work described above is a topic for continuing work.

The Unix program *bdiff* implements a sequence comparison algorithm for files that are too large to fit in RAM using a simple wrapper around the standard *diff* algorithm [Mye86]. The bdiff program works by first dividing the input files into segments that fit in RAM, and then running diff on each pair of corresponding (by position in the respective file) segments. This strategy does not guarantee a minimum-cost edit script. In fact, a single inserted line can mislead bdiff into mismatching a large number of other lines.

The problem of computing differences is closely related to the problem of pattern matching (e.g., [WZJS94, WCM+94, WSC+97]). While there are important differences between the two problems, it may be possible to share some of the techniques used by their solutions.

7 Conclusion

We described several applications that are based on comparing two snapshots of data in order to detect the differences between them. We explained the need for external-memory differencing algorithms for both flat (sequence) and hierarchical (tree) data. We modeled hierarchical data using rooted, ordered, labeled trees, and formalized the hierarchical data comparison problem using the idea of a minimum-cost edit script between two trees. We described a method to map this tree comparison problem to a shortest-path problem in a special graph called the tree edit graph. We first

presented a main-memory tree differencing algorithm based on the edit graph reduction. Next we discussed its extension to external memory and described problems with a naive extension. We then presented our main algorithm (xmdiff) for efficiently differencing trees in external memory.

As continuing work, we are exploring the use of our edit graph reduction to transfer results from sequence comparison to trees. Preliminary results indicate that while this strategy is relatively easy to use for main-memory algorithms, extending it to external-memory algorithms is more challenging. We are also working on incorporating the algorithm xmdiff into the C^3 change management system and on releasing a public version of the implementation. We also plan to explore the application of our techniques to related problems such as pattern matching and data mining in semistructured data (which is often modeled using trees and graphs).

References

[AHU76] 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, January 1976.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the International Conference on Data Engineering*, pages 4–13, Orlando, Florida, February 1998.

[CGM97] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[CRGMW96] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montréal, Québec, June 1996.

[LGM96] 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.

[MM85] W. Miller and E. Myers. A file comparison program. *Software–Practice and Experience*, 15(11):1025–1040, 1985.

[MP80] W. Masek and M. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[Mye86] E. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[Sel77] S. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[SK83] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[Tic85] W. Tichy. RCS—A system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[Vit98] J. Vitter. External memory algorithms. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Seattle, Washington, June 1998.

[WC76] C. Wong and A. Chandra. Bounds for the string editing problem. *Journal of the Association for Computing Machinery*, 23(1):13–16, January 1976.

[WCM+94] 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.

[WF74] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of the Association of Computing Machinery*, 21(1):168–173, January 1974.

[WMG90] S. Wu, U. Manber, and G.Myers. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317–323, September 1990.

[WSC+97] 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.

[WZJS94] 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, August 1994.

[Yan91] W. Yang. Identifying syntactic differences between two programs. *Software—Practice and Experience*, 21(7):739–755, July 1991.

[ZS89] 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.

[ZWS95] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 1995.