MANAGING CHANGE IN HETEROGENEOUS AUTONOMOUS DATABASES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Sudarshan Sudhir Chawathe
March 1999

Abstract

Information relevant to a task at hand is often scattered across a collection of heterogeneous, autonomous databases. Individual databases in such a collection are owned and managed by independent, and often competing, entities that cooperate to only a limited extent. For example, the collection of databases used in the construction of a building includes databases owned by the architect, the construction company, the electrical contractor, and so on. Such autonomous database collections are also common on the Internet. For example, the collection of Internet databases with information about San Francisco consists of databases operated by several competing entities. Making effective use of such collections of heterogeneous, autonomous databases presents several challenges due to the absence of traditional database facilities such as locks, transactions, and standard query languages. In particular, understanding and controlling how such databases evolve is an important problem that traditional database techniques are ill-equipped to address.

Managing evolving information in heterogeneous, autonomous databases requires (1) a method for detecting changes in data without access to traditional database control facilities such as triggers, transactions, and locks, and (2) a method for representing and querying these changes in a uniform manner. To address the first issue, we present efficient methods for detecting changes between snapshots of databases. Our methods are based on mapping the change detection problem to the problem of computing a concise representation of the difference between two labeled trees. We present the design and implementation of our algorithms for computing a concise difference between two trees, and study their performance both analytically and experimentally. An important distinguishing feature of our tree differencing algorithms is that they model changes using a rich set of edit operations. In addition to operations that insert and delete a node, and update node labels, our algorithms model subtree operations such as move and copy. Using a rich set of edit operations results in a more succinct and usable description of tree differences.

To address the second issue, we present a data model, DOEM, and query language, Chorel, for representing and querying changes in structured and semistructured data. A key feature of DOEM and Chorel is that they represent and query changes directly as first-class entities, instead of as the difference between database states. We describe how we use these ideas to implement CORE, a database system for historical semistructured data. We also describe the design and implementation of QSS, a service that supports subscriptions to interesting changes in heterogeneous, autonomous databases. QSS uses a powerful subscription language to specify the changes of interest.

Using the techniques of this dissertation, we have implemented the C^3 system for managing change in heterogeneous, autonomous databases. We describe the design and implementation of C^3 and our experiences with the system.

Dedicated to my parents, Sarala and Sudhir Chawathe.

Acknowledgements

My advisor, Hector Garcia-Molina, deserves the first thank you. Not only did Hector introduce me to interesting topics in the database field, he was also always willing to discuss my ideas, not matter how strange they were. From him, I have hopefully learned not only how to do research, but also how to interact effectively with students and colleagues. Jennifer Widom advised me on many technical topics throughout my Stanford years and provided a model for organization, planning, and balancing work with the rest of life. Serge Abiteboul helped me sort through my ideas on query languages and provided yet another great perspective on academic life. Thanks to Jeff Ullman for serving on my oral committee and for many fruitful discussions over the years. Gio Wiederhold, my academic grandfather, gave me the big picture on many issues, including how to drive a Volkswagen in Kanpur. The examples set by Hector, Jennifer, Serge, Jeff, and Gio played an important part in my decision to continue in academia.

I would like to thank all my other collaborators at Stanford. Anand Rajaraman was receptive to my idea of implementing a new difference program as a weekend project and made substantial contributions to the early work on *LaDiff*. (The project missed the time estimate by only a few years.) Vineet Gossain and Dan Liu made substantial contributions to the implementations of the *CORE* and *QSS* components of C^3. Thanks are also due to all members, past and present, of the *Lore* and *Tsimmis* projects.

The Stanford Database Group provided a great environment for both work and play. Among other things, I will miss the fine food and company at the Friday lunch meetings, the fiery discussions at the Thursday brownie meetings, and field trips that often involved sex (elephant seals at Año Nuevo) and violence (paint-ball in the Santa Cruz mountains). My officemates, Brad Adelberg and Yue Zhuge, deserve special mention for making sure I had something to do whenever I didn't feel like working. It's hard to imagine an office that's more fun. Ramana Yerneni joined our office when Brad left, and continued the fine tradition. Marianne Siroker was instrumental in making sure all official business got done without much effort on my part.

I was fortunate to meet some great people during my stay at Stanford. A special thank you to Amy McMullen for all the wonderful times. Thanks also go out to Venkat, Amy, Luca, Françoise, Jan, Dorothy, Cindy, and Melanie. The Stanford experience would have been much less enjoyable without friends like these. A number of institutions helped make sure I didn't get too carried away with work: TGIF, CoHo, Caffe Trieste, Cactus, F/X, SoFA, and Tahoe.

I dedicate this work to my parents, Sarala and Sudhir Chawathe, to whom I owe the greatest debt. I am also indebted to my sister, Supriya Pappu. I could not have come this far without their constant love, support, and sacrifices. Thanks to Ameya and Chamundeshwari for introducing me to the joys of being an uncle. Thanks are also due to the rest of my family, especially my grandparents. A special thank you goes out to my grandfather Manohar Goray.

This research was sponsored by by the Air Force Rome Laboratories under DARPA Contract F30602-95-C-0119, by the Air Force Wright Laboratory Aeronautical Systems Center under DARPA Contract F33615-93-1-1339, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The conclusions and opinions in this dissertation are those of the author and are not necessarily endorsed by any of these sponsors. The author makes no warranty of any kind, either expressed or implied, with regard to the programs and results contained in this dissertation, and shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of such material.

This document was typeset by the author in a paperless environment using TEX, LATEX, *xfig*, *Applix*, *ghostview*, and *emacs*.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are witnessing a proliferation of databases that are heterogeneous in their design and content, and that are operated by independent, often competing, organizations. Managing a collection of such heterogeneous, autonomous databases as a coherent information system necessitates a significant rethinking of several database techniques. In particular, managing the evolution of information stored in such a system is an important problem that is ill-addressed by conventional methods. In this dissertation, we present techniques for detecting, storing, querying, and monitoring changes in an environment of heterogeneous, autonomous databases.

We begin by introducing heterogeneous, autonomous databases in Section 1.1. Using an extended example, we describe how these databases differ from those studied in traditional database literature. We motivate the need for new database techniques by describing the dependence of traditional techniques on assumptions that are invalid for heterogeneous, autonomous databases. In Section 1.2, we briefly discuss the key research issues raised by the need for managing change in heterogeneous, autonomous databases. In Section 1.3, we present some examples of heterogeneous, autonomous databases from diverse application domains. Finally, Section 1.4 outlines the organization of the rest of this dissertation.

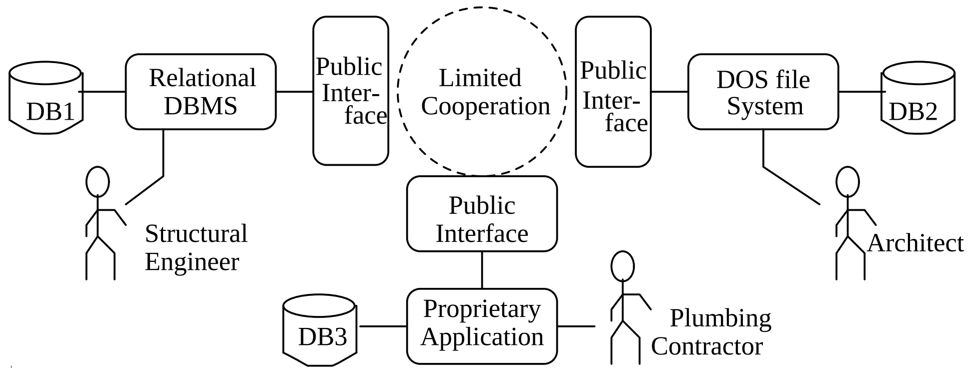


Figure 1.1: Heterogeneous, autonomous databases in the construction industry

1.1 Motivation

Traditional database research has focused on centralized database systems in which all data resides in a single database. More recent work has addressed parallel and distributed database systems, which store data in a collection of tightly coupled databases interconnected by a communication network. However, two key assumptions underlying these techniques are *homogeneity* and *centralized control* of the databases in the system. As a result, they are not applicable to *heterogeneous* and *autonomous* collections of databases. We elaborate on these terms using an example of such a collection.

Consider the collection of databases involved in the design and construction of a building. (Here, and in what follows, we use the term database to mean any organized collection of data. In addition to conventional relational and object databases, we include data from sources such as bibliographic information systems, file systems, world-wide web servers, and proprietary application systems.) A large number of independent parties are involved in the design and construction effort; Figure 1.1 depicts three such parties: the architect, the structural engineer, and the plumbing contractor. Each of these parties typically maintains one or more private databases.

Since these databases developed over time, in different organizations, and for differing goals, they are *heterogeneous*; that is, they differ widely in characteristics such as data models, query languages, access restrictions, and support for transactions, concurrency control, and locking. In our example, the structural engineer's database is stored in a relational database system, while the architect's database is simply a collection of files in a format used by a program for computer-aided design. The plumbing contractor's database is part of a proprietary application.

Traditional database techniques typically assume that the component databases in a distributed database system are *homogeneous*, making such techniques inapplicable to a heterogeneous database environment. In particular, such databases are assumed to be homogeneous in features such as the data models they support, the query language used to access their data, and the transaction and control primitives they support. Often, the assumptions of homogeneity are even more stringent. For example, many commercial products assume not only that the component databases are relational databases supporting the SQL2 query language [DD93], but also that they are identical versions of the same product from the same database system vendor.

Since the databases we study are typically owned by independent organizations, they are also *autonomous*; that is, they cooperate to only a limited extent, and do not expose sensitive or critical information to each other. Such database autonomy is most often motivated by business and legal reasons. Thus, even if facilities such as transactions, locks, and triggers exist in a database system (such as the structural engineer's relational database system in our example), they are typically not made available to the other databases in the collection. As shown in Figure 1.1, each database in the collection has a private interface that is available only to internal users, and a separate, and typically much more restricted, public interface that is presented to external users.

Prior work in distributed and federated databases typically assumes that the databases are *centrally administered*, making such work inapplicable in an autonomous environment. In particular, distributed or federated databases are assumed to be designed in a top-down fashion with the objective of supporting a group of applications efficiently. For example, techniques used for query processing in distributed databases assume that the data has been partitioned across the databases based on a careful analysis of the data, functional dependencies, and expected query mix. In contrast, the environment we study consists of databases that were not designed to facilitate interoperation. They are preexisting databases designed for differing purposes that we now wish to interconnect and use as a coherent system. In our ongoing construction example for instance, the architect's database is designed to facilitate and optimize the operations the architect is most likely to make, and may not support the operations of interest to a structural engineer.

Another reason traditional techniques are inapplicable to heterogeneous, autonomous databases is their assumption of a high level of trust amongst the component databases. The component databases are often required to perform a number of critical, and potentially dangerous, operations on behalf of each other. These operations include holding locks on data, exposing transaction commit states, and executing triggers. In an autonomous environment such sharing of critical resources between the component databases, which may belong to competing companies, is not plausible. Even if the owner of a database does not expect the owners of other databases to be malicious, the need to maintain organizational independence and accountability precludes sharing of critical database resources. For example, it is very unlikely that the structural engineer in our ongoing example would permit the architect to hold locks on the structural database, since such sharing of locks risks corrupting the structural database by factors beyond the structural engineer's control.

In addition to the differences in methods used by component databases to manage their data, there are also significant differences in the database contents themselves. The database contents are often mutually incompatible and inconsistent. For example, the plumbing contractor's database contains the locations, sizes, and types of pipes in the building; this information may be missing from the structural engineer's database. Similarly, the structural engineer's database may contain information on the physical properties of the beams and columns used in the building, but this information may be absent from the plumbing contractor's database. The architect's database may contain information about decorative features that is absent from the other databases. Further, since the architect may be working on a slightly newer version of the design than that used by the structural engineer, the two databases may disagree, for example, on the heights of some windows.

Despite all these differences among the component databases, a collection of heterogeneous, autonomous databases, such as the one in our ongoing construction example, represents a common reality, giving rise to the need to manage the collection as an *integrated information system*. For example, the final design of the building, as described by the collection of databases in our example, must be consistent. Thus, the location and thickness of a wall in the architect's database must be identical to the corresponding information in the structural engineer's database. In addition to this requirement of final consistency, the design databases of the parties involved in the construction need to be periodically synchronized with each other, and changes made by one party need to be propagated to affected parties in a timely manner. For example, if the architect modifies her database to reduce the clearance above a ceiling, the plumbing contractor may need to reroute pipes that no longer fit in the available space. Note that although such *consistency requirements* are similar to those found in traditional databases, there are important differences due to the autonomous environment in which the databases operate. Complete global consistency at all times is neither required nor practicable. We elaborate on these issues in Chapter 3.

1.2 Research Issues

In the above discussion, we motivated the need for a system to manage change in heterogeneous, autonomous databases. We also explained the reasons conventional database techniques cannot be used for this purpose. We now summarize the research issues raised by the design and implementation of a change management system for heterogeneous, autonomous databases. We present only a brief description of the issues here, with details deferred to Chapter 3.

Data Integration: Users of heterogeneous, autonomous database collections find it very cumbersome to learn and use the interface offered by each component database. They prefer a single, integrated interface to all the information in the database collection, irrespective of which database a particular data item resides in. The need to provide an integrated view over heterogeneous database collections raises several issues: First, we need a common data model that is general enough to encompass a wide variety of database types. Next, we need a method to translate queries over this general data model to queries on the underlying database, and similarly, to translate the results from the underlying database to the integrating model. Further, we need a query language that allows us to access and combine data from multiple sources, and methods for implementing and optimizing such queries. Although such *data integration* is not a focus of this dissertation, our work is designed to mesh well with such work, and our change management system makes use of data integration techniques in addition to the techniques of this dissertation.

Data that has been integrated from several diverse sources is typically *semi-structured*, meaning it has structure, but the structure may be irregular and incomplete, and may not conform to a fixed schema. This semistructured nature of the data in heterogeneous, autonomous databases introduces additional challenges in managing change in these databases. Most existing database techniques rely heavily on the existence of a stable and precise schema, and are thus inapplicable in a semistructured context.

Detecting Changes: A basic requirement of the change management system suggested in Section 1.1 is a method for *detecting changes* in heterogeneous, autonomous databases. Since we do not, in general, receive notifications of changes before or after they are made, we must use methods that detect changes by comparing snapshots of data. Although the problem of comparing data has been studied before, the characteristics of the data in a heterogeneous collection of databases pose challenges that require the development of new techniques for this purpose. We need data comparison techniques that can cope with the semistructured nature of the data, making effective use of the structure when available, but without assuming its presence in all cases. In Chapter 3, we describe how we map this problem to the problem of finding a concise description of the difference between two trees. In later chapters, we present the design, analysis, implementation, and experimental evaluation of tree differencing algorithms.

Representing and Querying Changes: Once we have detected changes using our differencing techniques, we need a method to systematically *store and query these changes*. Again, as a result of the semistructured nature of the data, we cannot use existing database techniques for this purpose. In this dissertation, we present a data model, called *DOEM*, for storing changes in semistructured data together with the data itself. We also present the design and implementation of a language, called *Chorel*, that allows us to query over historical semistructured data stored in a DOEM database.

Monitoring Changes: A system to manage change in heterogeneous, autonomous databases should include a facility for monitoring changes that are of interest. In order to implement such *subscriptions* to changes, we need a general-purpose language for specifying interesting changes. Further, we need techniques to implement subscriptions expressed in this language. The autonomy of the databases we consider makes these tasks particularly challenging.

1.3 Application Domains

Collections of heterogeneous, autonomous databases are becoming increasingly common. The principal reason for this increase is a proliferation of databases due to their falling costs. As the number of databases grows, administering them centrally quickly becomes impracticable. In Section 1.1, we presented an example of heterogeneous, autonomous databases from the domain of distributed design and construction. We now discuss a few other scenarios where such databases are found.

Consider the collection of databases found in large organizations such as multinational companies or major universities. These databases often number in the hundreds, and are designed, managed, and operated by relatively independent groups within the organizations. One group may operate a legacy IMS database, another may operate a modern relational database system, while a third group may use its own proprietary database system. For example, contact information for people in the Computer Science department at Stanford is stored in several separate databases, including a proprietary database designed and maintained by the department, a university-wide database maintained by the registrar's office, and the private databases of research, teaching, and recreational groups within the department. These databases differ widely in their data models, user interfaces, query facilities, reliability, and coverage. Currently, there is no systematic method used to manage the evolution of this collection of databases. Instead, people are expected to maintain consistent information in all these databases manually. As a result, these databases are often mutually inconsistent. The use of techniques in this dissertation would permit, for example, automatic notification when changes of a certain kind are made in one or more of these databases. Further, using DOEM and Chorel, one could query past states of these databases in order to generate a list of people whose database entries have been inconsistent for more than a week, so that these people could be notified to correct the situation.

Recent technical and market developments have led to an explosive growth in the number and variety of networked databases, especially on the World-Wide Web [BLCG92, W3C98]. The thousands of databases available on the Web are operated by independent, often competing, organizations, and vary widely in their design, data model, query facilities, accessibility, reliability, and consistency. Further, these databases rarely support the kinds of low-level access mechanisms required by traditional data management techniques. Using the techniques in this dissertation in this environment yields significant benefits. As a simple example, suppose we are interested in three Web sites. The first contains listings of show times in local movie theaters, the second contains movie reviews from a newspaper, and the third contains traffic reports from a television channel [PAW98, EG98, KRO98]. These three Web databases are operated by three separate organizations, and therefore exhibit a high degree of heterogeneity and autonomy. Further, the information contained in these databases changes frequently. Movie reviews and listings are updated once or twice a week, and traffic reports are updated every ten minutes. Monitoring and reacting to these changes is often of interest. For example, we may wish to be notified whenever a local theater adds a matinee show for any movie that has been received good reviews from the newspaper. Further, we may wish to be notified of any traffic problems near the theater. In Chapter 8, we describe a detailed example from this domain, illustrating how the techniques of this dissertation implement the desired functionality.

As another example, consider the increasingly common Web sites that sell books, compact discs, and other merchandise [AMA98, BN98, MBL98, CDN98, LND98]. Comparison shopping by visiting each of these Web sites is extremely tedious because such Web sites have vastly different interfaces. Often, these differences are intentional, since the parent organizations wish to distinguish themselves from their competitors. Using data integration techniques such as those developed in the *Tsimmis* project at Stanford [CGMH$^+$94], a convenient comparison shopping service can be implemented [JUN98]. Such a service asks a shopper for the desired product characteristics, such as the name of a book's author or the genre of music, and presents the shopper with a list of products that qualify, along with their prices and sources. Using the techniques in this dissertation, we can go even further. For example, if we are interested in a flat-panel computer monitor with a price less than $1000, but there are no such monitors currently on sale, we can set up a subscription that automatically monitors the relevant Web sites and notifies us when qualifying products become available. Our change management system, described in Chapter 8, supports such notifications and other, more complex, subscriptions to interesting changes in heterogeneous, autonomous databases.

1.4 Dissertation Organization

In Chapter 2, we discuss prior work in related fields. We focus on high-level similarities and differences between prior work and our work, deferring detailed comparisons to later chapters that present our techniques in detail. In Chapter 3, we present an overview of our framework for managing change in heterogeneous, autonomous databases. We present the conceptual architecture of the C^3 change management system, and identify its key modules and the subproblems they address. (The name C^3 suggests the three principal facets of change management: Changes, Configurations, and Consistency.)

Chapter 4 presents our techniques for detecting changes by comparing snapshots of data modeled using ordered trees. In this chapter, we model changes using node insertion, deletion and update, and subtree move operations. We present an algorithm that uses domain characteristics to yield efficient, optimal solutions for a large class of data. Our ability to model subtree moves in addition to the node operations used by prior work leads to a more compact, and intuitively more desirable, description of differences between trees. Some of the work in this chapter is reported in [CRGMW96].

Chapter 5 studies a similar change-detection problem for data that is modeled using unordered trees. In this chapter, we model changes using not only the earlier insert, delete, move, and update operations, but also subtree copy and uncopy operations. These additional operations allow us to describe changes more succinctly, and in a manner that is typically more useful to an application. We illustrate the benefits of our rich set of change operations, and describe the challenges in detecting such changes. We show how certain unintuitive descriptions of changes can be avoided by suitably restricting edit scripts, and present algorithms based on these ideas. Some of the work in this chapter is reported in [CGM97].

In Chapter 6, we present an alternative method of modeling transformations on tree-structured data. Instead of using a procedural description of changes, we introduce a declarative description that is roughly analogous to applying edit operations in parallel. This method of modeling transformations not only results in simpler change detection algorithms, but also produces change descriptions that are typically easier to understand compared to those that use the procedural model.

Chapter 7 describes how we store and query changes detected using the techniques from earlier chapters. We describe the OEM model for representing semistructured data, and the Lorel language for querying it. We then present our extension to OEM, called DOEM, that allows us to store the history of a semistructured database together with its content. We also present the syntax and semantics of our language, called Chorel, for querying semistructured data and its history. We describe how Chorel is implemented using a translation-based technique that allows us to take advantage of existing databases for semistructured and object data. Further, we illustrate the application of these ideas by describing the design and implementation of a service that supports subscriptions to interesting changes in heterogeneous, autonomous databases. Some of the work in this chapter is reported in [CAW98, CAW99].

In Chapter 8, we describe our implementation of the C^3 system for managing change in heterogeneous, autonomous databases. We describe the three major modules of our system: *TDiff*, which implements our differencing algorithms; *CORE*, which implements Chorel, and *QSS*, which implements subscriptions to changes. We first describe the facilities provided by C^3, and illustrate their use with the help of an extended example. We then describe how the C^3 modules interact with each other and with modules from related projects in order to implement these facilities. We also discuss the experiences we have had in using the C^3 system to monitor some databases on the Web.

In Chapter 9 we present the results of the experimental evaluation of our tree differencing algorithms. We study both the running time of our algorithms, and the quality of the solution they produce, presenting results for synthetically generated inputs as well as real inputs from the C^3 system. We conclude in Chapter 10 by summarizing the contributions of this dissertation and discussing promising directions for future work in related fields.

Chapter 2

Related Work

In this chapter we discuss prior work in topics related to this dissertation. In Section 2.1, we summarize work related to the problem of detecting changes by comparing snapshots of data, indicating how it differs from our work presented in Chapters 4–6 and 9. Section 2.2 discusses how prior work in temporal and hypothetical databases, and recent work on semistructured databases, relates to our design and implementation of a database system for historical semistructured data presented in Chapter 7. In Section 2.3, we briefly describe work in the field of data integration, indicating how our implementation of a change management system, described in Chapter 8, builds on this work. This chapter provides a high-level overview of related work; detailed comparisons with our techniques are found in the chapters describing those techniques.

2.1 Change Detection

The research literature contains a substantial body of work on the topic of comparing snapshots of data for detecting changes. We begin by discussing work on the problem of comparing strings and sequences, which has received the most attention in the literature. We then discuss work on the problem of comparing data that is more structured, such as data represented using ordered and unordered trees.

2.1.1 Strings and Sequences

Early interest in the problem of string comparison was motivated by applications such as spelling correction, and focused on relatively short strings (words). Later work has focused on comparing larger data for applications such as text comparison and version control. For example, [WF74] defines a string-to-string correction problem as the problem of finding the best sequence of insert, delete, and update operations that transform one string to another. The problem is developed further in [Wag75], which adds the "swap" operation to the list of edit operations. These papers also introduce the structure of a "trace" or a matching between the characters of the strings being compared as a useful tool for computing an edit script. A simpler change detection problem for strings, using only insertions and deletions as edit operations has been studied extensively [Mye86, Ukk85, WMG90]. The idea of a *longest common subsequence* (LCS) replaces the idea of a trace in this simpler problem.

A variant of the algorithm presented in [Mye86, Ukk85] for computing the longest common subsequence is implemented as the UNIX *diff* utility [HHS+98]. This *diff* program adds a number of features to the basic LCS algorithm to make it more usable. For example *diff* has options for grouping neighboring differences in *hunks*, ignoring whitespace and blank lines, ignoring the case of letters, and ignoring lines matching a specified regular expression. The *diff* program also includes a number of heuristics that improve performance at a small risk of producing a non-minimal solution. The output produced by *diff* can also be postprocessed in a variety of ways to make it more usable. For example, the *ediff* program highlights the differences computed by *diff* in the contexts of the two files being compared [Kif95]. It also selectively refines the differences by invoking *diff* on matching groups of lines to detect finer-grained differences which are then highlighted.

Given our goal of detecting changes in structured and semistructured data found in heterogeneous, autonomous databases, the biggest shortcoming of the work on comparing sequence data is that such algorithms do not take the hierarchical structure of the data into account. For example, when comparing documents, the structure imposed by paragraphs, sections, itemized lists, chapters, and so on, is ignored. Thus, a line in one file containing a section heading may be matched to a list item in the other, or a sentence may be matched to a sentence in a different paragraph although there is a reasonable match for it in its own paragraph. Furthermore, these algorithms cannot detect subtree operations such as moves and copies. Moves are reported as deletions and insertions, and copies simply as insertions. Thus, if a paragraph is moved from one section to another, it is reported as a deletion of some lines in the first section and an insertion of some lines in the other. Change detection facilities found in some application programs suffer from similar shortcomings. For example, Microsoft Word has a *revisions* feature that can detect simple updates, inserts, and deletes of text. However, it cannot detect moves or other subtree operations. WordPerfect has a *mark changes* facility that can detect some simple move operations. However, there are restrictions on how documents can be compared (on either a word, phrase, sentence, or paragraph basis). These approaches also do not generalize to non-document data. In brief, all the algorithms mentioned above work with strings or sequences, and are not suitable for computing changes in the structured and semistructured data found in the environments motivated in Chapter 1.

2.1.2 Ordered Trees

We can think of strings as ordered trees of height 1. When we consider more general ordered trees, the problem of detecting changes is more challenging than the string comparison problem because, intuitively, we need to find changes that account for not only the order among siblings, but also the ancestor relation. However, some simpler formulations of the ordered tree change detection problem can be solved efficiently. For example, if the only edit operations are insertions and deletions of subtrees, [Sel77] presents an efficient solution that is similar in spirit to the algorithm in [WF74]. Another formulation, using insertion, deletion, and label-update operations is studied in [ZS89], which presents a dynamic programming algorithm to solve the problem. The algorithm can be further improved if we assume all edit operations to have unit cost [SZ90].

Our algorithm for change detection in ordered trees, presented in Chapter 4, differs from prior work such as [ZS89] in three major ways: First, we use a different set of tree edit operations. In particular, in addition to node insertions, deletions, and label updates, we also permit subtree moves. As we will see in Chapter 4, subtree moves significantly improve the usability of the changes detected between trees, and also make the problem more challenging. Although we may attempt to detect moves using a postprocessing step, the results of such techniques can be far from optimal, especially when the number of differences is large [WZS95]. Further, in our work insertion and deletion operations operate only on leaf nodes, while in [ZS89] they are permitted to operate on interior nodes. The two sets of edit operations are equivalent in the sense that any state reachable using one set is also reachable using the other. The application domain usually determines which edit operations are more natural. In a general tree structure, the delete operation of [ZS89], which makes the children of the deleted node the children of its parent, is natural. However, in an object hierarchy, this may be undesirable due to restrictions on types and composite-object memberships. (For example, an object representing a library may have a number of book objects as subobjects. If a book is deleted, it is unnatural to have the subobjects of book (such as author, title, etc.) become subobjects of the library object.)

The second major difference between the work in Chapter 4 and prior work is that we make some assumptions about the nature of the data being represented. Our algorithm always yields correct results, but if the assumptions do not hold it may produce suboptimal results. Because of our assumptions, we are able to design an algorithm with a lower running-time complexity. In particular, our algorithm runs in time $O(ne + e^2)$, where n is the number of tree leaves and e is the "weighted edit distance" (typically, $e \ll n$). The algorithm in [ZS89] has time complexity $O(n^2 log^2 n)$ for balanced trees, and higher for unbalanced trees. The assumptions made by the algorithm in Chapter 4 are particularly well suited to documents and ordered semistructured data in formats such as HTML and XML commonly found on the Web [RHe98, BPSM98].

The third major difference between our change detection work in both Chapter 4 and 5 and prior work is a more subtle one: All prior work that we are aware of assumes that the function used to compare node labels satisfies the triangle inequality. That is, for any three labels a, b, and c, a label-comparison function f that returns

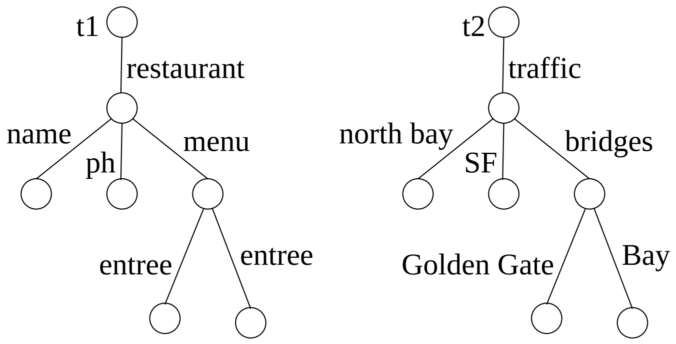


Figure 2.1: The need for flexible label comparison functions

the cost of updating one label to another must satisfy $f(a,c) \leq f(a,b) + f(b,c)$. Now this requirement alone may sometimes cause problems. For example, in an application merging two structured databases containing personnel records, we may wish to permit two ten-digit phone numbers to match provided either the first six digits (area code and exchange) match or the last seven digits (exchange and extension) match. If both these conditions are false, the comparison function returns a very high value to effectively prevent the nodes from matching. Thus we have the following situation that does not satisfy the triangle inequality: $f(415.723.0587, 415.723.6805) = 0.3$, $f(415.723.6805, 650.723.6805) = 0.1$, $f(415.723.0587, 650.723.6805) = 10$.

A more serious problem caused by the triangle inequality assumption of prior work is that most such work also requires the label comparison function to satisfy an extended form of the triangle inequality involving special labels $\oplus$ and $\ominus$. For notational convenience, insertion of a node with label l is often modeled as the edit operation $upd(\oplus, l)$; similarly, deletion is modeled using $upd(l, \ominus)$. The extended triangle inequality assumption then requires that $f(l_1, l_2) \leq f(l_1, \ominus) + f(\oplus, l_2)$; that is, the cost of updating label l_1 to l_2 cannot be greater than the cost of deleting a node with label l_1 and inserting a node with label l_2. In effect, techniques relying on this assumption do not offer any way for the application to indicate that certain nodes must never be matched to one another. For example, consider matching the two trees T_1 and T_2 that contain, respectively, the subtrees t_1 and t_2 suggested by

Figure 2.1. (This example is an abstraction of the *eGuide* database used in our implementation described in Chapter 8.) Given the semantics suggested by the labels of t_1 and t_2, it is clearly undesirable to match one to the other. However, if the two subtrees occur in similar positions in their respective trees, techniques that rely on the triangle inequality assumption will always match them to each other. In contrast, our techniques in Chapters 4, 5, and 6 allow us to use a more flexible label comparison function that assigns an arbitrarily high cost to updating the labels in t_1 to those in t_2.

Unfortunately, allowing a more flexible label-comparison function, while leading to more usable results, also makes it difficult to use traditional approaches based on dynamic programming to solve the change detection problem. Informally, dynamic programming solutions to the tree change detection problem use the following argument: Since the triangle inequality described above holds, it is not necessary to consider matching nodes that are "too far away." In particular, using the triangle inequality assumption, it is possible to derive results that constrain the kinds of node matchings that need to be considered. For example, it is often possible to rule out matchings that do not preserve the ancestor relation or the order among siblings [ZS89]. Without the triangle inequality assumption, we need to consider other techniques to simplify the problem; our techniques are presented in Chapters 4, 5, and 6.

2.1.3 Unordered Trees

The problem of detecting changes in unordered trees by computing a minimum-cost edit script that transforms one tree to another is inherently harder than the analogous problem for ordered trees. Even very simple formulations of this problem are extremely hard to solve. For example, [ZWS95] presents a proof of the $\mathcal{NP}$-hardness of a formulation that uses insertion, deletion, and label-update operations. The proof is by reduction from the *exact cover by three-sets problem*, which is known to be $\mathcal{NP}$-hard. Similar techniques can be used to show that most general formulations of this problem are hard.

Given the hardness of the problem, it seems necessary to explore techniques that rely on substantially restricting the problem, and heuristic techniques. An example of the former strategy is [ZWS95], which formulates a restricted version of the problem in which one can only insert and delete nodes with zero or one children. (This algorithm is also generalized to unrooted trees.) An example of the latter strategy is [SWZS94], which explores the effectiveness of standard search techniques such as probabilistic hill climbing.

As was the case for ordered trees, a major difference between prior work and our work presented in Chapters 4 and 5 is that we consider a much richer set of edit operations. In addition to the standard node insertion, deletion, and label update operations, we also permit subtree operations such as moves and copies. For example, when we compare documents, our techniques detect changes such as moved sections and copied paragraphs. Prior techniques detect such changes only as their component insertions and deletions, thus losing valuable information.

Another distinguishing feature of our work, also similar to the case for ordered trees, is that we do not insist that the function used to compare labels satisfy the triangle inequality. Note that when we consider unordered trees, assuming the triangle inequality does not lead to any significant simplification of the problem. In particular, the hardness result mentioned above still holds.

In general, there are several formulations of the problem of detecting changes in snapshots of data. In addition to being useful for managing change in autonomous databases, such techniques have applications in many other domains. For example, [Yan91] describes the application of a technique similar to that in [Sel77] to identify syntactic differences between versions of a program. The formulation that is most profitable to use depends on the application at hand. In an application with a small amount of data (e.g., structured catalogue entries), or when we are willing to spend more time (e.g., biochemical structures), more thorough search algorithms may be preferred. However, in applications with large amounts of data (e.g., object hierarchies, database dumps), or with strict running-time requirements, we would use our algorithm. The efficiency of our method is based on exploiting certain domain characteristics. Even in domains where these characteristics may not hold for all of the data, it may be preferable to get a quick, correct, but not guaranteed optimal, solution using our approach. The variations we have explored in this dissertation are those we have found well-suited to our purpose of managing change in autonomous databases.

2.2 Representing and Querying Changes

Consider the general problem of representing and querying the history of a database in addition to its current state. Prior work on this topic takes one of the following two approaches. The first approach, which we call the *snapshot-collection* approach, models the history of a database as a collection of database states, or snapshots. In this model, a change operation takes a database from one state to the next. The states are ordered, usually linearly, based on some parameter, usually time or version number. In addition to permitting queries on the current database state, this model permits any other state of the database to be queried. This approach is used by temporal databases [SA86, Soo91]. The second approach, which we call the *snapshot-delta* approach, models the history of the database using a single database snapshot and a collection of *deltas*. In this model, we obtain various states of the database by starting with a single snapshot and applying some sequence of deltas to it. We use the snapshot-delta approach in our work. An early, simple example of this approach is the use of *delta relations* in active databases and trigger languages [Buc96, WC96a, Mel96]. In such work, changes to a relation R are represented using two relations, R^+ and R^-, where $R^+ = R_{new} - R_{old}$, and $R^- = R_{old} - R_{new}$. More recently, this approach has been used by work on hypothetical database systems, which permit queries over database states obtained by applying a set of deltas to the current state [GHJ96, DHR96, GH97].

The traditional approach to representing and querying changes in a database models changes to only the content of a database, not its schema, which is assumed to be fixed. That is, only those database changes that are consistent with the schema are modeled. For example, in relational databases such techniques model the insertion, deletion, and update of tuples in a relation. They do not model other kinds of changes to the database, such as creation and removal of relations, addition and removal of relation attributes, and changes in key constraints. Similarly, for object databases, traditional techniques model only changes that are consistent with the schema. For example, changes such as modifying a class by adding or removing a data member, defining a new class, and changing the subtype hierarchy are not modeled. Therefore, traditional techniques for representing change in a database cannot model any change that causes an explicit (by design of the administrator) or implicit (as a result of new data with different characteristics) change in the schema.

2.2.1 Heterogeneous Databases and Schemas

The reliance of traditional work on a fixed schema causes serious difficulties when working with heterogeneous, autonomous databases. First, designing a relational or object schema for the data found in heterogeneous databases is extremely difficult. For example, consider trying to design a schema for the Web site of a newspaper [NYT98, WP98]. These databases are more similar to structured documents than they are to traditional databases, and they rarely adhere to a strict code of presentation and semantics. Intuitively, the reason for the difficulties in modeling heterogeneous databases using a schema is the following: Every schema relies on a set of assumptions. For example, relational database schema design is guided by the presence and absence of functional dependencies [Arm74, Ull88]. Heterogeneous databases by their very nature lack the consistency, stability, and structure implied by these assumptions. These difficulties are exacerbated when the data of interest comes from not one but several databases that have been integrated because, as the number of integrated databases grows, the likelihood of any assumption being valid for all the databases drops sharply. Further, even if we are fortunate enough to find a schema that works for the collection of heterogeneous databases under consideration, there is no guarantee that we will not soon encounter new data that does not conform to this schema, since any assumptions the schema makes are not guaranteed to hold in the source data. (This problem is analogous to the problems one may expect if functional dependencies in a relational database are guessed by observing only the current state of the database. Although the current state of the database may not contradict the guessed dependencies, it is likely that a future database state will.) As a result of these difficulties in schema design for heterogeneous databases, many systems that work with such data are forced to use a degenerate schema of the form $(id, type, value)$, effectively reducing a database system to a storage system. Using such a degenerate schema results in complex queries for even simple retrieval tasks and adversely affects the performance of the database system.

The difficulties caused by changes in schema have been noted in earlier work, and techniques have been proposed for managing *schema evolution*. For example, the ORION object database system includes facilities that allow the class hierarchy defining the database schema to be modified [BKKK87]. However, an important assumption made by such work is that schema changes are made only rarely (e.g., once a year). Consequently, such techniques often rely on some manual intervention and perform a significant amount of database restructuring for each schema change. Therefore, these techniques are not useful if the schema is expected to change more rapidly, as is the case for heterogeneous databases.

2.2.2 Semistructured Databases

One may observe that the above arguments about the disadvantages of relying on a fixed schema when working with heterogeneous databases are valid even if we are interested in modeling a simple, non-historical database. Indeed, similar observations have led to recent work on the topic of *semistructured databases* [AQM+96, MAG+97, BDHS96, Abi97]. Informally, semistructured databases are databases that have some structure; however, this structure is irregular, incomplete, and subject to frequent changes. Such databases are often described as schema-less. In contrast to conventional database systems that first define a schema and then populate the database, semistructured database systems first populate a database using a very general data model and then try to infer and use the regularities in the data. The greatest advantage of using such a semistructured data model with heterogeneous databases is that we do not need to perform the difficult and tedious task of designing and maintaining an integrating schema.

The disadvantage of a semistructured data model is that the implementation of common database system functions becomes more challenging. For example, the tasks of data layout, query processing and optimization, and indexing can no longer rely on a fixed schema and therefore require new techniques. There has been recent preliminary work on such topics. For example, techniques for inferring structure and regularity in semistructured databases are studied in [NUWC97, GW97], and [MW98] describes the application of query optimization techniques to the *Lorel* language for querying semistructured data [AQM^{+}96]. However, much work remains to be done in the emerging topic of semistructured databases. In particular, representing and querying changes in semistructured databases is significantly more challenging than it is in databases with a fixed schema. To our knowledge, our work presented in Chapter 7 is the first to address this important problem.

We now summarize the major differences between our work presented in Chapter 7 and prior work. The first difference is the data model used. Most prior work uses the relational data model in which there is a simple notion of changes: tuples in a relation may be inserted, deleted, or updated. Such changes are modeled using one of the two approaches (snapshot-collection and snapshot-delta) mentioned above. However, these are not the only changes that can be made to a relational database. We can also create new relations, destroy existing ones, modify the definitions of relations, add or remove key constraints, and so on. Such changes, which are ignored by traditional work on representing and querying changes in relational databases, are captured by our work.

Another major difference between our work and prior work is that we treat changes as first-class entities, not only in data representation, but also in our query language. Prior work, such as [DHR96], adopts the following strategy for querying a historical or hypothetical database: Some subset of the changes represented in the database are selected and applied to the current state of the database, producing another (historical or hypothetical) state. This state is then queried in the standard manner. In contrast, our query language presented in Chapter 7 allows a finer-grained mixing of the application of changes and querying.

Yet another distinguishing feature of our work is that our data model and query language use very few primitives. In addition to simplifying the implementation and use of our system, a smaller number of primitives imposes fewer requirements on the kinds of data we can represent. For example, it is extremely tedious to coerce data obtained from the Web into a regular structure, relational or object based. Even with advances in standards such as XML, the structure of the data remains unreliable and fluid. Thus, using a simple graph-based model that makes very few assumptions about the structure of the data being modeled has significant advantages.

2.3 Data Integration

Although data integration is not the focus of this dissertation, our framework for managing change in heterogeneous, autonomous databases, described in Chapter 3, includes modules that rely on data integration techniques. In particular, our implementation of the C^3 change management system, described in Chapter 8, uses modules from the *Tsimmis* project on data integration [CGMH$^+$94, LYV$^+$98].

The goal of data integration is to shield a casual user of multiple heterogeneous databases from the intricacies of the differences in data models, query languages, and access methods supported by these databases. For example, an integrated view over two personnel databases, one relational and the other object-oriented, presents all employee records in the same manner irrespective of the database to which an individual record belongs. One can find, for example, employees hired in the past year by posing a single query over the integrated view without worrying about details of the underlying relational and object query languages and schemas.

We need to address the problem of data integration at several levels, ranging from simple, syntactic integration to complex, semantic integration. Although the boundary between syntactic and semantic integration depends on the representation used, it is useful to intuitively position integration tasks on a continuum ranging from tasks requiring only simple translation to those requiring complex reasoning. For example, the task of integrating design databases that store measurements using different units (e.g., inches and centimeters) requires only simple translation. Integrating product catalogs that list prices using different currencies is a more difficult task, since the conversion rates between currencies depend on a number of complex factors. As an example of difficult semantic integration, consider the task of integrating the accounting databases of two companies. This task requires a detailed understanding of the accounting principles used by the companies in order to design a suitable mapping between the databases. For instance, a person considered as a consultant by one company may be treated as an employee by another. Thus, answering even a simple query asking for the names of employees in the two companies is difficult.

Given the complexity of semantic integration, an integration strategy that uses only fully automated techniques is not likely to succeed. Most recent work in data integration has therefore focused on designing techniques that facilitate and partially automate the task of integrating diverse sources. A common strategy is to use modules called *wrappers* to translate data from the data model used by a source database to the data model used for integration [HBGM+97]. Further integration is achieved using *mediators*, which are modules that interact with wrappers and other mediators in order to support an integrated view of data from multiple sources [Wie92]. The work described in this dissertation builds on this framework of wrappers and mediators used for data integration. Chapters 3 and 8 describe how the C^3 system interacts with wrappers and mediators.

We may classify data integration techniques into two broad categories: lazy and eager [Wid96]. The *lazy* approach, which we also call the *virtual integration* approach, performs query translation, query execution, and data translation and integration only when it is needed to execute a query. As described above, this approach uses wrappers and mediators to integrate data from diverse sources. The Tsimmis project has developed methods for rapid implementation of wrappers and mediators. Wrappers are generated by specifying query and data translation using a high-level language based on pattern-matching [PGGMU95, HGMC+97, HBGM+97]. Similarly, mediators are generated from high-level specifications using a language that is similar to Datalog [PGMU96, PAGM96, LYV+98, UW97]. Similar techniques for integrating data using wrappers and mediators have been developed in several other projects, including Rufus [SLS+93], Garlic [CHS+95], SIMS [ACHK93], Pegasus [ADD+94], and the Information Manifold [KLSS95].

In contrast to the lazy approach to data integration, the *eager* approach, which we also call the *materialized integration* or *data warehousing* approach, captures all the data of interest from the source databases in advance of any query execution. This data is translated, integrated, and stored in a central database called the warehouse. User queries over the integrated view are then answered by simply querying the warehouse. This approach is taken by the *Whips* project at Stanford [HGMW$^+$95] and other data warehousing projects [Inm92]. The task of translating and integrating data using the eager approach is quite similar to that using the lazy approach. Materialized integration uses modules called *grabbers* to extract data from the source databases. Many of the wrapper implementation techniques based on pattern-matching are applicable to grabbers. However, unlike wrappers, grabbers extract data when the warehouse is set up, without waiting for any queries.

The lazy and eager approaches to integration have the advantages and disadvantages that are characteristic of lazy and eager strategies in general. For example, the lazy approach avoids performing work not required for query execution, while the eager approach permits faster and more reliable query execution after initial warehouse set-up. The eager approach also requires techniques for keeping the integrated data stored in the warehouse up-to-date. This problem is similar to the materialized view maintenance problem [BLT86]; however, the heterogeneity and autonomy of the source databases introduce additional complications and there has been recent work to address these issues [ZGMHW95, HZ96]. As we will describe in Chapters 3 and 8, the C^3 system uses a combination of the virtual and materialized integration approaches.

Chapter 3

Overview

In this chapter, we present a brief overview of our strategy for managing change in heterogeneous, autonomous databases. We sketch the architecture of the C^3 change management system and briefly describe its key components. The details of the techniques used to implement these components are deferred to later chapters. (As indicated in Chapter 1, whe name C^3 suggests the three principal facets of change management: Changes, Configurations, and Consistency.) Recall from Chapter 1 that throughout this dissertation we use the term *databases* to denote heterogeneous, autonomous collections of data. In addition to the well studied relational and object-oriented database systems, we use the term databases to denote collections of unstructured or semistructured data in various data formats. For example, we include data stored in formats such as plain text, HTML, XML, SGML, ASN.1, Bibtex, Refer and MIF, and data that is accessible through protocols such as SMTP, FTP, NNTP, HTTP, Finger, and WHOIS [RHe98, BPSM98, Gol90, Uni93, Lam94, Pos82, PR85, KL86, FGM+97, Zim90, HSF85]. Our interest lies in managing a collection of such diverse databases as a coherent information system; in particular, we are interested in managing change in these databases. In what follows, we refer to the heterogeneous, autonomous databases that we wish to manage as the *source databases.*

3.1 Integrating Heterogeneous Databases

Given the heterogeneity in the data models, query languages, and access methods of the source databases, we first need a strategy to avoid the proliferation of special techniques needed to interact with each different kind of database. We would like to present the users of our system a unified view of the data in all the source databases, irrespective of the characteristics of the source database from which a particular data item is obtained. For this purpose, we need an *integrating data model* that is simple and general enough to encompass a wide variety of source data models. As discussed in Chapter 2, *semistructured* data models are particularly well suited to this purpose. We use the *Object Exchange Model (OEM)*, which was devised as part of the *Tsimmis* project, as our integrating model [PGMW95, CGMH+94]. In OEM, a database is simply a rooted, labeled, directed graph. Nodes in this graph have labels denoting data content, and arcs have labels denoting the relationship between the nodes they connect.

We would also like to present our users with a single query language to query over the data integrated from all the source databases, irrespective of the particular query languages supported by the databases containing data relevant to the query. We use the *Lorel* query language, designed as part of the *Lore* project to query over integrated data represented in OEM [AQM+96, MAG+97]. The central idea in Lorel is the use of general path expressions. These are sequences of labels, including optional wildcards and regular expressions, that intuitively match certain paths in the OEM graph. (We describe OEM and Lorel in detail in Chapter 7.) Thus our strategy is to support a uniform abstraction, in OEM, of the source databases, and to support Lorel queries over this abstraction.

In order to implement the above strategy for accessing heterogeneous databases using OEM and Lorel, we need the ability to translate Lorel queries to the native query languages of the source databases, and to translate the results of the native query (in the native data format) to OEM. The modules that implement the above functionality are called *wrappers*. Wrapper implementation techniques have been studied in several works, including [PGGMU95, AK97, HGMC+97, HBGM+97], and

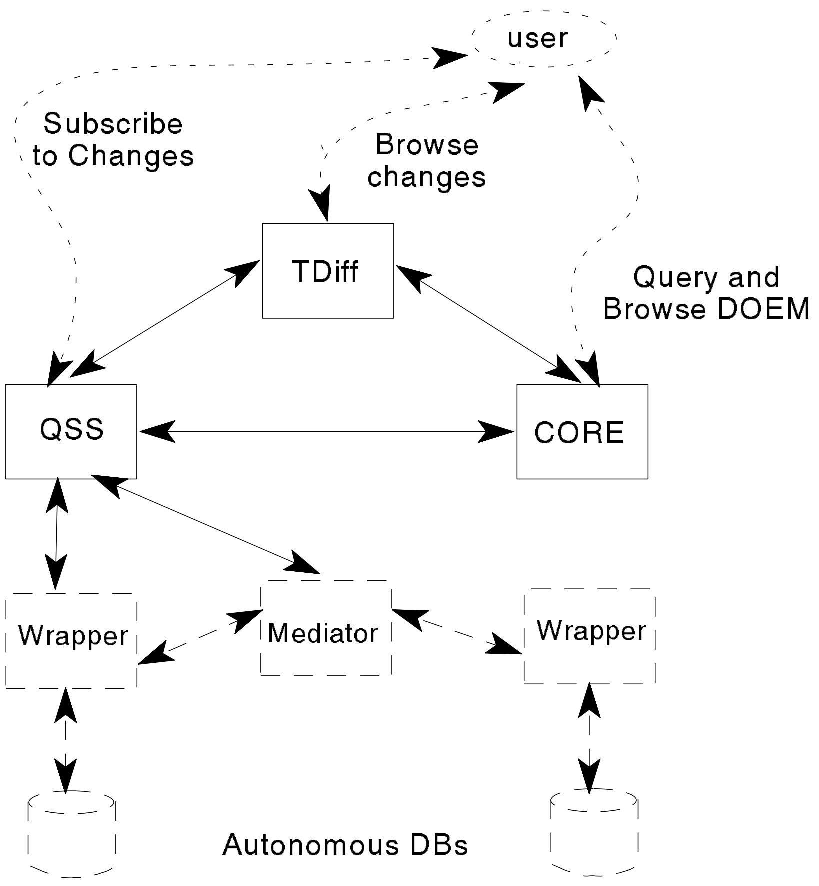


Figure 3.1: Conceptual architecture of the C^3 system

we do not describe them in this dissertation. Briefly, there is one wrapper for each source database. This wrapper accepts a Lorel query, and translates it into a suitable query in the native query language of the source database. Often, the translated query is one that returns a superset of the desired results since the exact query may not be supported by the source. The native query is executed by sending it to the source database, and the results are filtered if needed, and translated to OEM. In order to combine data from multiple heterogeneous databases, we use *mediators*, which are modules that interact with wrappers and other mediators [Wie92, PGMU96]. An example of a simple mediator is a fusion mediator that combines data from two or more databases [PAGM96]. Wrappers and mediators thus provide the rest of the system with a simple abstraction of source databases: Source databases accept a Lorel query and return OEM results.

3.2 Detecting Changes

In order to manage changes in the source databases, we must first detect them. The autonomy and heterogeneity of these databases necessitates special techniques for this purpose. Consider the comparison depicted in Figure 3.2. In a traditional database system, the only way to access the database is through the database management system (DBMS). In particular, the database can be modified only through the DBMS. Thus detecting changes to the database is a matter of simple bookkeeping. In contrast, an autonomous database system may or may not be stored using a DBMS. Even if a database uses a DBMS, the DBMS interface is private and accessible only to the owner of the database (for autonomy reasons). An external user of such a database can access the database only through a restricted public interface that accepts queries and returns results. As discussed above, a data integration system typically accesses such databases through a wrapper. Thus we need techniques to detect changes in the source databases using only the wrapper interface.

In some cases, a source database provides some additional functionality that is useful for detecting changes. For example, a Web site listing books for sale may offer a feature to notify users when books of a certain type arrive. Such trigger and

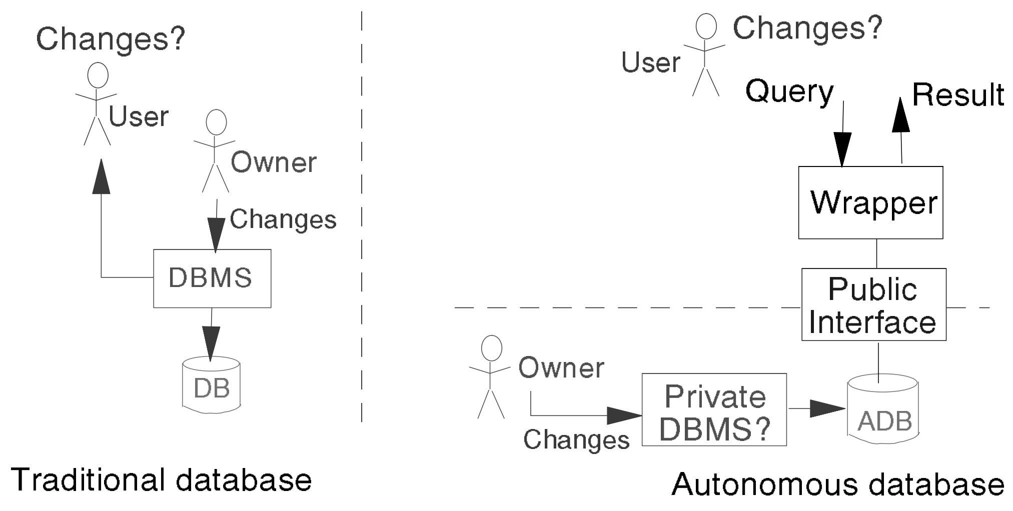
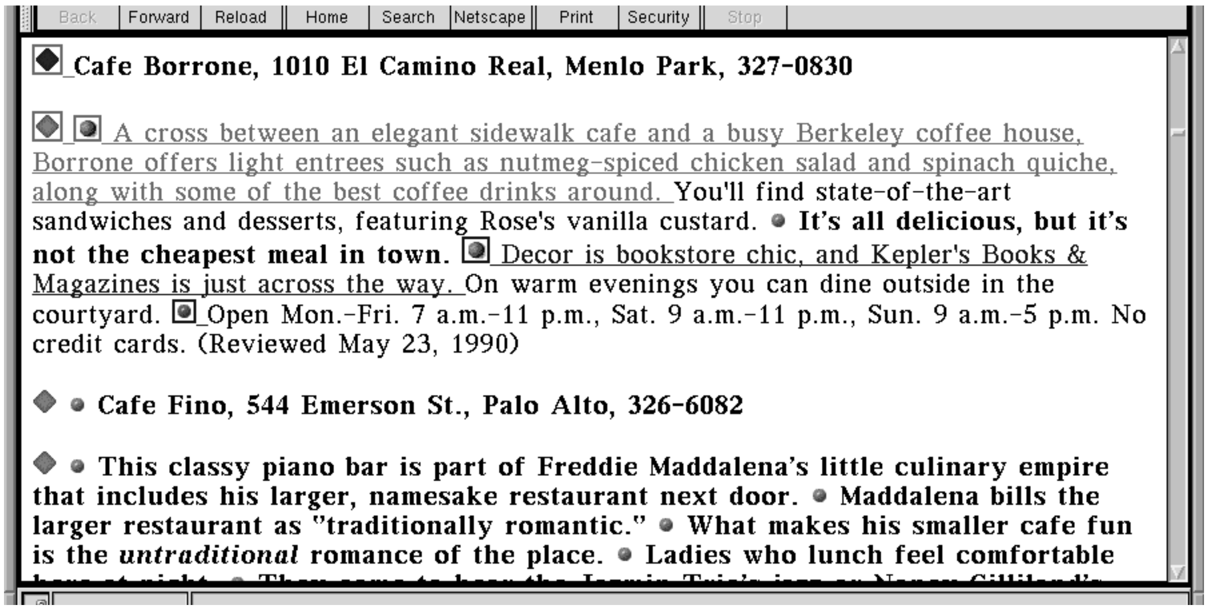


Figure 3.2: Changes in autonomous databases

notification mechanisms, when supported, can be used to aid the change detection process. However, in general we cannot rely on the existence of such facilities, and need a method for detecting changes based only on the query interface provided by wrappers.

Thus, in order to detect changes in the source databases, we need to poll these databases and detect changes by comparing the old and new data snapshots. More precisely, we periodically send a query to the wrapper of a source database, resulting in a sequence of query results over time. We compare each pair of successive results and detect any differences between them. The query results returned by Tsimmis wrappers are tree structured. Therefore, we need a method to compare two snapshots of tree structured data and detect changes between them. In order to make this tree comparison more precise, we need to define the type of trees we consider, the edit operations used to modify them, the manner in which such edit operations are applied, and the properties desired of the detected changes. For example, a trivial way to describe the difference between any two trees is to indicate that all the nodes in one tree are deleted, followed by the insertion of the nodes in the second tree. Although technically correct, such a description is not very useful if, for example, the two trees



◆ **Cafe Borrone, 1010 El Camino Real, Menlo Park, 327-0830**

◆ ● A cross between an elegant sidewalk cafe and a busy Berkeley coffee house, Borrone offers light entrees such as nutmeg-spiced chicken salad and spinach quiche, along with some of the best coffee drinks around. You'll find state-of-the-art sandwiches and desserts, featuring Rose's vanilla custard. ● **It's all delicious, but it's not the cheapest meal in town.** ● Decor is bookstore chic, and Kepler's Books & Magazines is just across the way. On warm evenings you can dine outside in the courtyard. ● Open Mon.-Fri. 7 a.m.-11 p.m., Sat. 9 a.m.-11 p.m., Sun. 9 a.m.-5 p.m. No credit cards. (Reviewed May 23, 1990)

◆ ● **Cafe Fino, 544 Emerson St., Palo Alto, 326-6082**

◆ ● **This classy piano bar is part of Freddie Maddalena's little culinary empire that includes his larger, namesake restaurant next door.** ● **Maddalena bills the larger restaurant as "traditionally romantic."** ● **What makes his smaller cafe fun is the *untraditional* romance of the place.** ● **Ladies who lunch feel comfortable**

Figure 3.3: Sample output from TDiff

differ only in a single node. In Chapters 4, 5, and 6, we present different formulations of the tree differencing problem, and present techniques for solving them.

Using our tree differencing techniques, we have implemented the *TDiff* module of the C^3 system. This module takes two data snapshots as input and presents as output a concise description of the differences between them. We have also developed a graphical interface that presents the computed differences as mark-up on the input data. For example, a version of TDiff specialized for HTML data takes two Web pages as input and produces as output a marked-up HTML document in which inserted, deleted, updated, and moved textual units are indicated using icons. Figure 3.3 shows an excerpt from the output of TDiff on two versions of a Web page listing restaurant reviews from the Palo Alto Weekly [PAW98]. Each icon represents an edit operation, with the color indicating the type of the operation, and the shape indicating the textual unit (sentence, paragraph, or section). Clicking on an icon reveals more information about the edit operation. For example, clicking on a red dot, which signifies a deleted sentence, results in the display of a marked-up copy of the old version of the document, with the deleted sentence highlighted. We describe our implementation in more detail in Chapter 8. Of course, generating a specialized version of TDiff for every new kind of source data we encounter is not practical. Instead, as described above, the C^3 system uses wrappers to translate such data into OEM, and a single version of TDiff that operates on OEM data.

3.3 Managing Changes

The graphical interfaces to the C^3 system allow us to browse the changes between two OEM snapshots. However, just as browsing a database is not practical once the size of the database grows beyond a few kilobytes, browsing changes becomes impractical when we are dealing with large amounts of data. We need a method for systematically storing and querying these changes. Note that we need the ability to store and query changes over several versions of the data, not just the two most recent ones. In particular, since we use OEM as our integrating data model, we need techniques to represent, store, and query historical OEM data, that is, OEM data and the changes made to it over time. In Chapter 7 we present our extension to OEM that allows us to model data and changes in a simple and general manner. We call this extension *DOEM*, for Delta-OEM. DOEM is well suited to browsing marked-up versions of OEM data using a method similar to the one in Figure 3.3 for HTML data. In addition, DOEM is well suited to querying using a general-purpose, powerful, query language. We have designed a query language, called *Chorel*, for historical data represented in DOEM. Chapter 7 describes the syntax, semantics, and implementation of Chorel. Chorel extends Lorel to permit querying not only the current state of a semistructured database, but also its history of changes. For example, we can write a query over the history of a Web database with movie listings to find horror movies that began showing in one theater within one week of when they stopped showing in another [EG98]. Using the techniques in Chapter 7, we have implemented a database system for historical semistructured data. This database system is called *CORE*, for Change Object Repository, and is implemented by extending the Lore database system for OEM data. Thus changes detected by our TDiff module are stored in a DOEM database in the CORE module, which permits browsing and querying.

The last major component of the C^3 change management system is *QSS*, which stands for *Query Subscription Service.* Using the TDiff and CORE modules, QSS supports subscriptions to changes in the source databases. For example, a QSS subscription over a Web site with movie listings may request that it be notified whenever a new action movie starts showing at a theater in Palo Alto. The syntax, semantics, and implementation of such subscriptions are described in Chapter 7. The QSS module also acts as a driver for the rest of the C^3 system. It polls the source databases at appropriate times by sending a query to the wrapper, sends the new and previous result to TDiff, and installs any changes detected by TDiff in the appropriate DOEM database in CORE. Further, it executes Chorel queries over the DOEM databases stored in CORE in order to detect changes that satisfy some subscription, and sends any such changes to the subscription owner. Our implementation of the C^3 system is described in more detail in Chapter 8.

3.4 Summary

In this chapter, we outlined our high-level strategy for addressing the problem of managing change in heterogeneous, autonomous databases. We presented the conceptual architecture of the C^3 system, including the modules we use from the related projects Tsimmis and Lore. In order to limit the amount of special-purpose design required due to the heterogeneity of source databases, we use a simple graph-based integrating model called OEM, and conceptually map all data to this format. We use template-based wrappers from the Tsimmis project to translate queries and data between the OEM model and the models used by the source databases. Further integration is achieved using a network of mediators that support powerful mechanisms to combine and transform data. Although the design and implementation of wrappers and mediators, along with other topics in data integration, are interesting research issues, they are not the focus of this dissertation; therefore we do not discuss them in detail.

Since we cannot assume the availability of sophisticated notification facilities, detecting changes in autonomous databases requires techniques to compute changes by comparing snapshots of data. We briefly described how we formalize these ideas by defining a tree differencing problem. In Chapters 4, 5, and 6, we study in detail the formulation and solution of such tree differencing problems.

Since we use a semistructured data model, OEM, as our integrating model, the tasks of representing, storing, and querying the changes detected by our differencing techniques present some unique challenges. We have developed a data model, DOEM, and a query language, Chorel, for historical semistructured data to address these challenges. We present DOEM and Chorel in Chapter 7, which also describes a powerful subscription service called QSS. An important feature of our work on representing and querying changes is the treatment of changes as first class concepts.

In this chapter, we described only briefly the manner in which the work to be described in later chapters contributes to the task of building a change management system for heterogeneous, autonomous databases. We present the details in Chapter 8, which describes the C^3 system from both user and implementor standpoints.

Chapter 4

Detecting Changes in Ordered Trees

In Chapter 3 we described the high-level architecture of our change management system, and noted that a key component of our system is a module that detects changes by comparing snapshots of structured or semistructured data. In the next three chapters, we describe techniques for detecting changes in this manner. In this chapter, we focus on algorithms for detecting changes in data that is modeled using layered, ordered trees. We formalize the change detection problem as the problem of computing a minimum-cost edit script that transforms the tree modeling one snapshot to the tree modeling the other. In order to detect changes that are more meaningful to an application or end user, we permit our edit scripts to contain not only the traditional node insertion, deletion, and update operations, but also operations that move entire subtrees. We use domain characteristics to simplify the problem, and present an efficient algorithm that is optimal for data with these characteristics. We introduce a convenient representation of an edit script as a *delta tree*, and describe our implementation of a differencing program based on these ideas. In summary, the main contributions of this chapter are the following:

- a formal definition of the problem of detecting changes in structured and semi-structured data given the old and new versions of the data;

- efficient algorithms for computing a minimum cost edit script between two trees;
- analytical and empirical performance studies of our algorithms;
- a general scheme, called a delta tree, to represent changes in hierarchically structured information.
- a powerful *LaDiff* system for detecting and representing changes in hierarchically structured Latex documents that demonstrates the utility of our approach.

The remainder of this chapter is organized as follows. Section 4.1 describes our general approach, divides our problem into two distinct subproblems, and provides preliminary definitions. Our algorithms for solving the two subproblems are discussed in Sections 4.2 and 4.3. Section 4.4 describes delta trees. In Section 4.5 we describe the application of our techniques to hierarchically structured documents. Our empirical performance study is described in Section 4.6, and are followed by a chapter summary in Section 4.7.

4.1 Overview

In this section, we formulate the change detection problem and split it into the following two subproblems which are discussed in later sections:

- Finding a "good" matching between the nodes of the two trees;
- Finding a minimum "conforming" edit script for the two trees given a computed matching.

We first introduce these problems informally using an example. The formal definitions follow in Section 4.1.1, which also introduces some notation and terms used in the rest of the chapter.

Recall, from Section 7.1, that we wish to detect changes between snapshots of data represented using *ordered trees*—trees in which the children of each node have a designated order. Hereafter, when we use the term "tree" we mean an ordered tree. We consider trees in which each node has a *label* and a *value*. These trees are

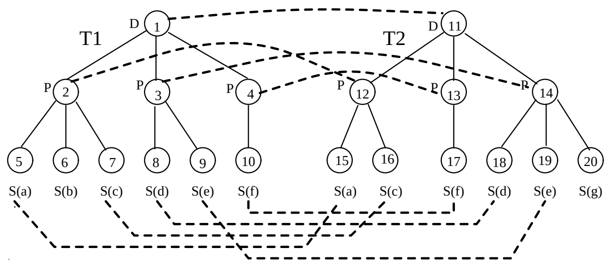


Figure 4.1: Running example (dashed edges represent matching)

natural abstractions of the OEM data model that we briefly introduced in Chapter 3. (Details of how OEM data is mapped to ordered trees are in Chapter 7.) We also assume that each tree node has a unique identifier; identifiers may be generated by our algorithms when they are not provided in the data itself. Note that the nodes that represent the same real-world entity in different versions may not have the same identifier. We refer to the node with identifier x as "node x" for conciseness.

As a running example, consider trees T_1 and T_2 shown in Figure 4.1, and ignore the dashed lines for the moment. The number inside each node is the node's identifier and the letter beside each node is its label. All of the interior nodes have null values, not shown. Leaf nodes have the values indicated in parentheses. (These trees could represent two structured documents, where the labels `D`, `P`, and `S` denote `Document`, `Paragraph`, and `Sentence`, respectively. The values of the sentence nodes are the sentences themselves.) We are interested in finding the delta between these two trees. We will assume that T_1 represents the "old" data and T_2 the "new" data, so we want to determine an appropriate transformation from tree T_1 to tree T_2.

Our first task in finding such a transformation is to determine nodes in the two trees that correspond to one another. Intuitively, these are nodes that either remain unchanged or have their value updated in the transformation from T_1 to T_2 (rather than, say, deleting the old node and inserting a new one). For example, node 5 in T_1 has the same value as node 15 in T_2, so nodes 5 and 15 should probably correspond. Similarly, nodes 4 and 13 have one child node each, and the child nodes have the same value, so nodes 4 and 13 should probably correspond. The notion of a correspondence between nodes that have identical or similar values is formalized as a *matching* between node identifiers. Matchings are one-to-one. We say that a matching is *partial* if only some nodes in the two trees participate, while a matching is *total* if all nodes participate. Hereafter, we use the term "matching" to mean a partial matching unless stated otherwise.

Hence, one of our problems is to find an appropriate matching for the trees we are comparing. We call this problem the *Good Matching* problem. In some application domains the Good Matching problem is easy, such as when data objects contain object identifiers or unique keys. In other domains, such as structured documents, the matching is based on labels and values only, so the Good Matching problem is more difficult. Furthermore, not only do we want to match nodes that are identical (with respect to the labels and values of the nodes and their children), but we also want to match nodes that are "approximately equal." For instance, node 3 in Figure 4.1 probably should match node 14 even though node 3 is missing one of the children of 14. Details of the Good Matching problem—including what constitutes a "good" matching—are addressed in Section 4.3. A matching for our running example is illustrated by the dashed lines in Figure 4.1.

We say that two trees are *isomorphic* if they are identical except for node identifiers. For trees T_1 and T_2, once we have found a good (partial) matching M, our next step is to find a sequence of "change operations" that transforms tree T_1 into a tree T_1' that is isomorphic to T_2. Changes may include inserting (leaf) nodes, deleting (leaf) nodes, updating the values of nodes, and moving nodes along with their subtrees. Intuitively, as T_1 is transformed into T_1', the partial matching M is extended into a total matching M' between the nodes of T_1' and T_2. The total matching M' then defines the isomorphism between trees T_1' and T_2. We call the sequence of change operations an *edit script*, and we say that the edit script *conforms* to the original matching M provided that $M' \supseteq M$. (As will be seen, an edit script conforms to partial matching M as long as the script does not insert or delete nodes participating in M.) Edit scripts are defined in more detail shortly.

We would like our edit script to transform tree T_1 as little as possible in order to obtain a tree isomorphic to T_2. To capture minimality of transformations, we introduce the notion of the *cost* of an edit script, and we look for a script of minimum cost. Thus, our second main problem is the problem of finding such a minimum cost edit script; we refer to this as the *Minimum Conforming Edit Script (MCES)* problem. The remainder of this section formally defines edit operations and edit scripts. Our algorithm for the MCES problem is presented in Section 4.2, and Section 4.3 presents our algorithm for the Good Matching problem. Note that we consider the MCES problem before the Good Matching problem, despite the fact that our method requires finding a matching before generating an edit script. As will be seen, the definition of a good matching relies on certain aspects of edit scripts, so for presentation purposes we consider the details of our edit script algorithms first.

4.1.1 Edit Operations, Edit Scripts, and Costs

We now formalize the concepts we introduced informally above. We define the operations we use for editing trees, describe how a sequence of such edit operations is used to transform a tree, and define the cost of such a sequence of edit operations.

Edit Operations

In an ordered tree, if nodes $v_1, \ldots, v_m$ are the children of node u, then we call v_i the ith child of u. For a node x, we let $l(x)$ denote the label of x, $v(x)$ denote the value of x, and $p(x)$ denote the parent of x if x is not the root. We assume that labels are chosen from a fixed but arbitrary set. In the definitions of the edit operations, T_1 refers to the tree on which the operation is applied, while T_2 refers to the resulting tree. The four edit operations on trees are the following:

Insert: The *insertion* of a new leaf node x into T_1, denoted by $\textsc{ins}((x, l, v), y, k)$. A node x with label l and value v is inserted as the kth child of node y of T_1. More precisely, if $u_1, \ldots, u_m$ are the children of y in T_1, then $1 \leq k \leq m+1$ and $u_1, \ldots, u_{k-1}, x, u_k, \ldots, u_m$ are the children of y in T_2. The value v is optional, and is assumed to be null if omitted.

Delete: The *deletion* of a leaf node x of T_1, denoted by $del(x)$. The result T_2 is the same as T_1, except that it does not contain node x. $del(x)$ does not change the

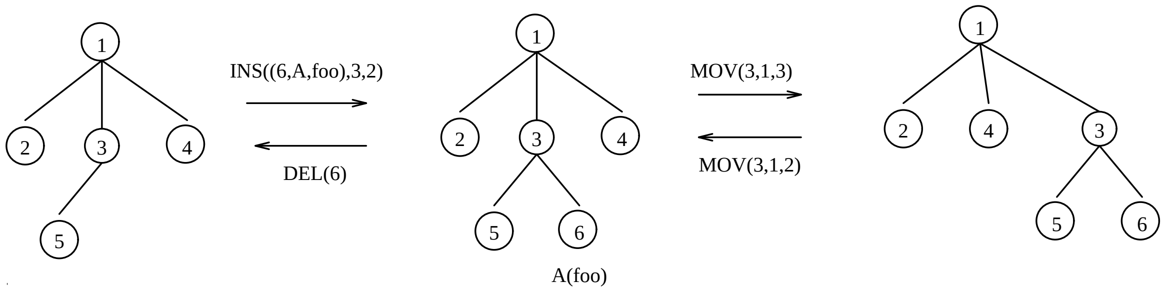


Figure 4.2: Edit operations on a tree

relative ordering of the remaining children of $p(x)$. This operation deletes only a leaf node; to delete an interior node, we must first move its descendents to their new locations or delete them.

Update: The *update* of the value of a node x in T_1, denoted by $upd(x, val)$. T_2 is the same as T_1 except that in T_2, $v(x) = val$.

Move: The *move* of a subtree from one parent to another in T_1, denoted by $\textsc{mov}(x, y, k)$. T_2 is the same as T_1, except x becomes the kth child of y. The entire subtree rooted at x is moved along with x.

Figure 4.2 shows examples of edit operations on trees. In the figure, node 6 has label A and value *foo*. The labels and values of the other nodes are not shown.

Edit Scripts

Informally, an edit script gives a sequence of edit operations that transforms one tree into another. Formally, we say $T_1 \xrightarrow{e_1} T_2$ when T_2 is the result of applying the edit operation e_1 to T_1. Given a sequence $E = e_1, \ldots, e_m$ of edit operations, we say $T_1 \xrightarrow{E} T_{m+1}$ if there exist $T_2, \ldots, T_m$ such that $T_1 \xrightarrow{e_1} T_2 \xrightarrow{e_2} \ldots \xrightarrow{e_m} T_{m+1}$. A sequence E of edit operations *transforms* T_1 into T_2 if $T_1 \xrightarrow{E} T_1'$ and T_1' is isomorphic to T_2. (Recall that two trees are isomorphic if they differ only in the identifiers of their nodes.) We call such a sequence of edit operations an *edit script of* T_1 *with respect to* T_2. Notice that an edit script does not tell us how the original matching between

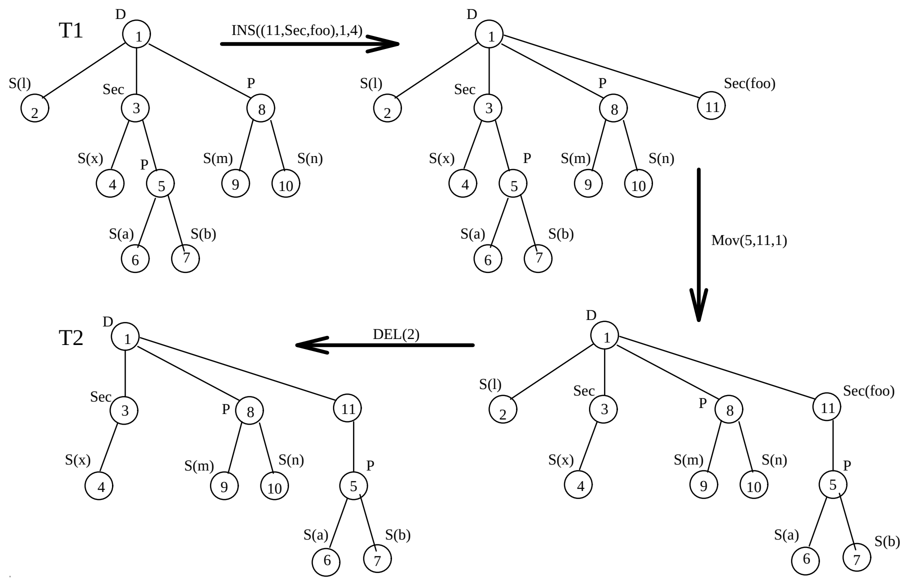


Figure 4.3: Applying the edit script of Example 4.1.1

T_1 and T_2 should be modified to obtain the total matching between T_1' and T_2. This will be done as the edit script is generated; see Section 4.2.

Example 4.1.1 Consider the trees T_1 and T_2 shown in Figure 4.3. The following edit script below transforms T_1 into T_2:

$$\text{INS}((11, \mathit{Sec}, \mathit{foo}), 1, 4), \text{MOV}(5, 11, 1), \mathit{del}(2), \mathit{upd}(9, \mathit{baz})$$

Figure 4.3 also shows the intermediate trees in the transformation specified by the above edit script. (The last update is not shown in order to save space.)

A Cost Model for Edit Scripts

Given two trees, in general there are many edit scripts that transform one tree to the other. Even when an edit script must conform to a given matching, there may be many correct scripts. (Recall that we defined the concept of an edit script conforming to a matching in Section 4.1.) For example, the following edit script, when applied to the initial tree in Example 4.1.1, produces the same final tree as that produced by the edit script in the example:

$$\text{INS}((11, \mathit{Sec}, \mathit{foo}), 1, 4), \mathit{del}(6), \mathit{del}(7), \mathit{del}(5),$$
$$\text{INS}((12, S, a), 11, 1), \text{INS}((13, S, b), 11, 2), \mathit{upd}(9, \mathit{baz})$$

Intuitively, this edit script does more work than necessary, and is thus an undesirable representation of the delta between the trees. To formalize this idea, we introduce the *cost* of an edit script.

We first define the costs of edit operations and then use these costs to define the cost of edit scripts. The cost of an edit operation depends on the type of operation and the nodes involved in the operation. Let $c_D(x)$, $c_I(x)$, and $c_U(x)$ denote respectively the cost of deleting, inserting, and updating node x, and let $c_M(x)$ denote the cost of moving the subtree rooted at node x. In general, these costs may depend on the label and the value of x, as well as its position in the tree. In this chapter, we adopt a simple cost model where deleting and inserting a node, as well as moving a subtree, are considered to be unit cost operations. That is, $c_D(x) = c_I(x) = c_M(x) = 1$ for all x.

Now consider the cost $c_U(x)$ of updating the value of a node x. We assume that this cost is given by a function, *compare*, that evaluates how different x's old value v is from its new value v'. This *compare* function takes two nodes as arguments and returns a number in the range $[0, 2]$. Although the nature of the *compare* function is arbitrary, it should be consistent with the costs of the other edit operations in the following sense: Suppose x is moved, and its value v is updated so that v is very similar to v'. Then $\mathit{compare}(v, v')$ should be less than 1, so that the cost of moving and updating x is less than the cost of deleting x and replacing it with a new node with value v'. If v and v' are very different, we would rather have the edit script contain a delete/insert pair, so the update cost should be greater than 1. Finally, the cost of an edit script is the sum of the costs of its individual operations. We can now state our problem succinctly as follows:

Problem Definition: Given two trees T_1 and T_2, find a minimum-cost edit script that transforms T_1 into T_2.

We solve this problem in two steps:

1. We find a (partial) matching M between the nodes of T_1 and T_2.
2. We then find a *Minimum Conforming Edit Script (MCES)* for T_1, T_2 and M; that is, an edit script that conforms to M, and that transforms T_1 into T_2 such that there is no other edit script conforming to M that transforms T_1 into T_2 and has a lower cost.

Of course, this two-step procedure will result in the desired minimum-cost edit script only if we select an appropriate matching in the first step. We discuss how that is done in Section 4.3. For presentation purposes, it is more convenient to discuss the second step first, in the next section.

4.2 Generating the Edit Script

In this section we describe how we solve the *Minimum Conforming Edit Script* problem motivated in the previous section: Given a tree T_1 (the *old tree*), a tree T_2 (the *new tree*), and a (partial) matching M between their nodes, generate a minimum cost edit script that conforms to M and transforms T_1 to T_2. Our algorithm starts with an empty edit script E and appends edit operations to E as it proceeds. To explain the working of the algorithm, we apply each edit operation to T_1 as it is added to E. When the algorithm terminates, we will have transformed T_1 into a tree that is isomorphic to T_2. In addition, the algorithm extends the given partial matching M by adding new pairs of nodes to M as it adds operations to E. When the algorithm terminates, M is a total matching between the nodes of T_1 and T_2.

4.2.1 Outline of Algorithm

The algorithm is most easily described as consisting of five phases: the *update phase*, the *align phase*, the *insert phase*, the *move phase*, and the *delete phase*. We describe each phase in turn. Let us call a node that is not matched in M an *unmatched node*. The *partner* of a matched node is the node to which it is matched in M. We use our running example from Figure 4.1. We are required to find a minimum cost edit script that transforms T_1 into T_2, given the matching M shown by the dashed lines in the figure.

The Update Phase: In the update phase, we look for pairs of nodes $(x, y) \in M$ such that the values at nodes x and y differ. For each such pair (in any order) we add the edit operation $upd(x, v(y))$ to E (recall that for a node x, $v(x)$ denotes the value of x), and we apply the update operation to T_1. At the end of the update phase, we have transformed T_1 such that $v(x) = v(y)$ for every pair of nodes $(x, y) \in M$.

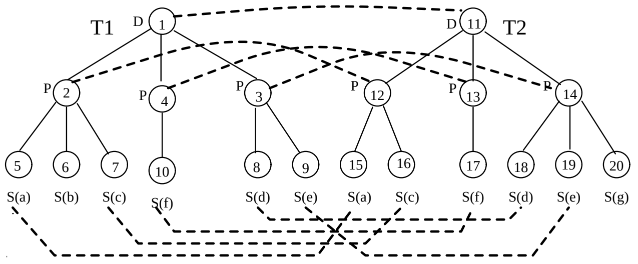


Figure 4.4: Running example: after align phase

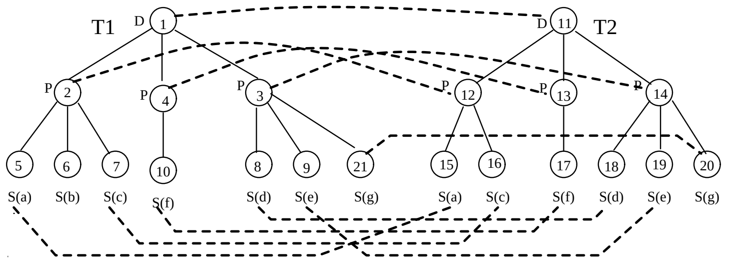


Figure 4.5: Running example: after insert phase

The Align Phase: Let the *partner* of a node denote the node to which it is matched (by a given matching). Suppose $(x, y) \in M$. We say that the children of x and y are *misaligned* if x has matched children u and v such that u is to the left of v in T_1 but the partner of u is to the right of the partner of v in T_2. In Figure 4.1, the children of the root nodes 1 and 11 are misaligned. In the align phase we check each pair of matched internal nodes $(x, y) \in M$ (in any order) to see if their children are misaligned. If we find that the children are misaligned, we append move operations to E to align the children. We explain how the move operations are determined in Section 4.2.2 below. In our running example, we append $\text{MOV}(4, 1, 2)$ to E, and we apply the move operation to T_1. The new T_1 is shown in Figure 4.4.

The Insert Phase: We assume, without loss of generality, that the roots of T_1 and T_2 are matched in M. (If the roots of T_1 and T_2 are not matched in M, then we add new dummy roots that are matched.) In the insert phase, we look for an unmatched node $z \in T_2$ such that its parent is matched. Suppose $y = p(z)$ (i.e., y is the parent of z) and y's partner in T_1 is x. We create a new identifier w and append $\text{INS}((w, l(z), v(z)), x, k)$ to E. The position k is determined with respect to the children of x and z that have already been aligned with respect to each other; details are in Section 4.2.3. We also apply the insert operation to T_1 and add (w, z) to M. In our running example we append $\text{INS}((21, S, g), 3, 3)$. The transformed T_1 and the augmented M are shown in Figure 4.5. At the end of the insert phase, every node in T_2 is matched but there may still be nodes in T_1 that are unmatched.

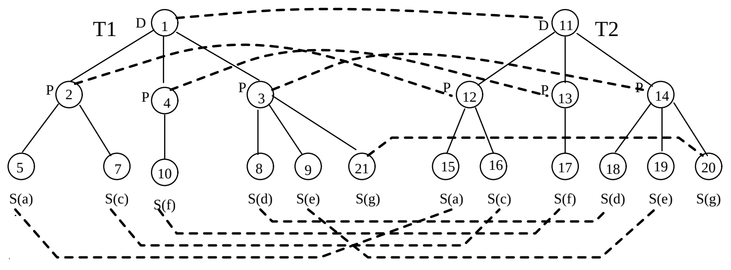


Figure 4.6: Running example: after delete phase

The Move Phase: In the move phase we look for pairs of nodes $(x, y) \in M$ such that $(p(x), p(y)) \notin M$. (Recall from Section 4.1.1 that $p(x)$ denotes the parent of x.) Suppose $v = p(y)$. We know that at the end of the insert phase, v has some partner u in T_1. We append the operation $\text{MOV}(x, u, k)$ to E, and we apply the move operation to T_1. Here the position k is determined with respect to the children of u and v that have already been aligned, as in the insert phase. At the end of the move phase T_1 is

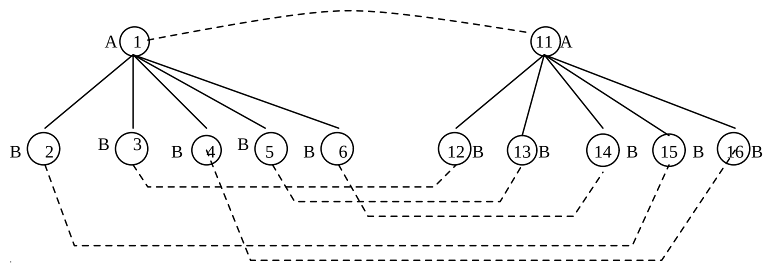


Figure 4.7: A matching with misaligned nodes

isomorphic to T_2 except for unmatched nodes in T_1. In our running example, we do not need to perform any actions in this phase.

The Delete Phase: In the delete phase we look for unmatched leaf nodes $x \in T_1$. For each such node we append $del(x)$ to E and apply the delete operation to T_1. (Note that this process will result in a bottom-up delete—descendents will be deleted before their ancestors.) At the end of the delete phase T_1 is isomorphic to T_2, E is the final edit script, and M is the total matching to which E conforms. Figure 4.6 shows the trees and the matching after the delete phase.

4.2.2 Aligning Children

The Problem: The align phase of the edit script algorithm presents an interesting problem. Suppose we detect that for $(x, y) \in M$, the children of x and y are misaligned. In general, there is more than one sequence of moves that will align the children. For instance, in Figure 4.7 there are at least two ways to align the children of nodes 1 and 11. The first consists of moving nodes 2 and 4 to the right of node 6, and the second consists of moving nodes 3, 5, and 6 to the left of node 2. Both yield the same final configuration, but the first one is better since it involves fewer moves.

To ensure that the edit script generated by the algorithm is of minimum cost, we must find the shortest sequence of moves to align the children of x and y. Our algorithm for finding the shortest sequence of moves is based on the notion of a *longest common subsequence*, described next.

Longest Common Subsequence: Given a sequence $S = a_1a_2\ldots a_n$, a sequence

S' is a *subsequence* of S if it can be obtained by deleting zero or more elements from S. That is, $S' = a_{i_1} \ldots a_{i_m}$ where $1 \leq i_1 < i_2 < \ldots < i_m \leq n$. Given two sequences S_1 and S_2, a *longest common subsequence (LCS)* of S_1 and S_2, denoted by $LCS(S_1, S_2)$, is a sequence $S = (x_1, y_1) \ldots (x_k, y_k)$ of pairs of elements such that

1. $x_1 \ldots x_k$ is a subsequence of S_1;

2. $y_1 \ldots y_k$ is a subsequence of S_2;

3. for $1 \leq i \leq k$, $equal(x_i, y_i)$ is true for some predefined equality function $equal$; and

4. there is no sequence S' that satisfies conditions 1, 2, and 3 and is longer than S.

The length of an LCS of S_1 and S_2 is denoted by $|LCS(S_1, S2)|$. □

We use an algorithm due to Myers [Mye86] that computes an LCS of two sequences in time $O(ND)$, where $N = |S_1| + |S_2|$ and $D = N - 2|LCS(S_1, S_2)|$. We treat Myers' LCS algorithm as having three inputs: the two sequences S_1 and S_2 to be compared, and an equality function $equal(x, y)$ used to compare $x \in S_1$ and $y \in S_2$ for equality. That is, we treat it as the procedure $LCS(S_1, S_2, equal)$.

The Solution: The solution to the alignment problem is now straightforward. Compute an LCS S of the matched children of nodes x and y, using the equality function $equal(u, v)$ that is true if and only if $(u, v) \in M$. Leave the children of x that are in S fixed, and move the remaining matched children of x to the correct positions relative to the already aligned children. In Figure 4.7, the LCS is $3, 5, 6$ (matching the sequence $12, 13, 14$). The moves generated are MOV$(2, 1, 5)$ and MOV$(4, 1, 5)$. Lemma 1 below shows that our LCS-based strategy always leads to the minimum number of moves.

Lemma 1 *For sequences S_1 and S_2 and an equality function* equal *such that each element in S_1 is equal to exactly one element in S_2 and vice versa, the minimum number of moves of elements of S_1 required to align the elements of S_1 and S_2 is* $|S_1| - |\mathrm{LCS}(S_1, S_2)|$. □

```
1. $E \leftarrow \epsilon$, $M' \leftarrow M$
2. Visit the nodes of $T_2$ in breadth-first order
   /* this traversal combines the update, insert, align, and move phases */
   (a) Let $x$ be the current node in the breadth-first search of $T_2$ and let $y = p(x)$. Let $z$ be
       the partner of $y$ in $M'$. (*)
   (b) If $x$ has no partner in $M'$
       i. $k \leftarrow FindPos(x)$
       ii. Append $\textsc{ins}((w, a, v(x)), z, k)$ to $E$, for a new identifier $w$.
       iii. Add $(w, x)$ to $M'$ and apply $\textsc{ins}((w, a, v(x)), z, k)$ to $T_1$.
   (c) else if $x$ is not the root /* $x$ has a partner in $M'$ */
       i. Let $w$ be the partner of $x$ in $M'$, and let $v = p(w)$ in $T_1$.
       ii. If $v(w) \neq v(x)$
           A. Append $upd(w, v(x))$ to $E$.
           B. Apply $upd(w, v(x))$ to $T_1$.
       iii. If $(y, v) \notin M'$
           A. Let $z$ be the partner of $y$ in $M'$. (*)
           B. $k \leftarrow FindPos(x)$
           C. Append $\textsc{mov}(w, z, k)$ to $E$.
           D. Apply $\textsc{mov}(w, z, k)$ to $T_1$.
   (d) $AlignChildren(w, x)$
3. Do a post-order traversal of $T_1$. /* this is the delete phase */
   (a) Let $w$ be the current node in the post-order traversal of $T_1$.
   (b) If $w$ has no partner in $M'$ then append $del(w)$ to $E$ and apply $del(w)$ to $T_1$.
4. $E$ is a minimum cost edit script, $M'$ is a total matching, and $T_1$ is isomorphic to $T_2$.
```

Figure 4.8: Algorithm *EditScript*

Proof. Suppose we can use fewer moves. Then consider the elements of S_1 that were not moved and their "partners" in S_2. They would form a common subsequence longer than $|LCS(S_1, S_2)|$, a contradiction. □

4.2.3 The Complete Algorithm

We now present the complete algorithm to compute a minimum cost edit script E conforming to a given matching M between trees T_1 and T_2. In the algorithm, we combine the first four phases of Section 4.2.1 (the update, insert, align, and move phases) into one breadth-first scan on T_2. The delete phase requires a post-order

```
Function AlignChildren(w, x)
  1. Mark all children of $w$ and all children of $x$ "out of order."
  2. Let $S_1$ be the sequence of children of $w$ whose partners are children of $x$ and let $S_2$ be the
     sequence of children of $x$ whose partners are children of $w$.
  3. Define the function $equal(a, b)$ to be true if and only if $(a, b) \in M'$.
  4. Let $S \leftarrow LCS(S_1, S_2, equal)$.
  5. For each $(a, b) \in S$, mark nodes $a$ and $b$ "in order."
  6. For each $a \in S_1$, $b \in S_2$ such that $(a, b) \in M$ but $(a, b) \notin S$
     (a) $k \leftarrow FindPos(b)$
     (b) Append $\textsc{mov}(a, w, k)$ to $E$ and apply $\textsc{mov}(a, w, k)$ to $T_1$.
     (c) Mark $a$ and $b$ "in order."

Function FindPos(x)
  1. Let $y = p(x)$ in $T_2$ and let $w$ be the partner of $x$ $(x \in T_1)$.
  2. If $x$ is the leftmost child of $y$ that is marked "in order," return 1.
  3. Find $v \in T_2$ where $v$ is the rightmost sibling of $x$ that is to the left of $x$ and is marked "in
     order."
  4. Let $u$ be the partner of $v$ in $T_1$.
  5. Suppose $u$ is the $i$th child of its parent (counting from left to right) that is marked "in order."
     Return $i + 1$.
```

Figure 4.9: Functions *AlignChildren* and *FindPos* used by Algorithm *EditScript*

traversal of T_1 (which visits each node after visiting all its children). The order in which the nodes are visited and the edit operations are generated is crucial to the correctness of the algorithm. (For example, an insert may need to precede a move, if the moved node becomes the child of the inserted node.) The algorithm applies the edit operations to T_1 as they are appended to the edit script E. When the algorithm terminates, T_1 is isomorphic to T_2. The algorithm also uses a matching M' that is initially M, and adds matches to it so that M' is a total matching when the algorithm terminates.

The algorithm is shown in Figure 4.8. It uses two procedures, *AlignChildren* and *FindPos*, shown in Figure 4.9. The two statements in Algorithm *EditScript* that are marked with (*) claim that certain nodes have partners. These claims are substantiated in the proof of the following theorem about the correctness and running time of our algorithm:

Theorem 1 *Algorithm EditScript computes the minimum cost edit script that conforms to the given matching M, and it does so in time $O(ND)$ where N is the number of nodes in the two trees and D is the number of misaligned nodes. (Typically D is much smaller than N.)* $\square$

Proof. We first show that the edit script E that is generated transforms T_1 to T_2 and conforms to M. The proof is in two stages.

In the first stage we show that at the end of the breadth-first traversal of T_2, the subtree of T_1 corresponding to only its matched nodes (under M') is isomorphic to T_2. The proof is by induction on the number of nodes visited so far by the breadth-first search. The induction hypothesis is the following: Consider the subtree T_2^s of T_2 that contains only nodes that have already been visited by the breadth-first search. Let T_1^s be the subtree of T_1 that contains only partners of the nodes in T_2^s. Then T_1^s is isomorphic to T_2^s. Moreover, every node in T_2^s is matched to some node in T_1^s in M'. The details of the induction are straightforward and are omitted.

In the second stage we show that the post-order traversal of T_1 deletes all the unmatched nodes in T_1, so that T_1 becomes isomorphic to T_2. The only problem we may face is that some node that we wish to delete has children and so the deletion is not a legal operation. Suppose some unmatched nodes in T_1 are not deleted. Let x be a "lowest" such node in T_1, i.e., a node that occurs before all other such nodes in the post-order numbering. Then it follows from the first part of the proof that x does not have any children in T_1. Hence x could have been deleted during the post-order traversal of T_1, a contradiction.

Thus E transforms T_1 to T_2. It is also clear that E conforms to M because E never deletes any nodes that are matched by M. We also note that the inductive proof used in the first stage shows that the claims made by the statements marked with a (*) in Algorithm *EditScript* are indeed correct.

We now show that E is a minimum cost edit script. Any edit script conforming to M must contain at least:

- one insert operation corresponding to each unmatched node in T_2;

- one delete operation corresponding to each unmatched node in T_1; and
- one move operation corresponding to each pair of matched nodes $(x, y) \in M$ such that $(p(x), p(y)) \notin M$ (call these *inter-parent moves*).

It is clear that Algorithm *EditScript* generates precisely the above inserts, deletes, and inter-parent moves. All that remains is to show that the algorithm also generates the fewest possible *intra-parent* moves (moves that change the relative ordering of siblings). Such moves are generated only in Function *AlignChildren*. That the minimum possible number of such moves is generated is an immediate consequence of Lemma 1. Hence E is a minimum cost edit script.

We first define the notion of *misaligned nodes*. Suppose $x \in T_1$ and $y = p(x)$. A move of the form $M(x, y, k)$ for some k is called an *intra-parent move* of node x; such moves are generated in the align phase of the algorithm. The number of misaligned nodes of T_1 with respect to T_2 is the minimum number of intra-parent moves among all minimum cost edit scripts. Other than in Function *AlignChildren*, the breadth-first search and post-order traversal perform a constant amount of work for each node in T_1 and T_2. Let $|x|$ denote the number of children of node x. For matched nodes $w \in T_1$ and $x \in T_2$, let $d(x, w)$ denote the number of misaligned children of x and w. Then Function *AlignChildren* aligns the children of w and x in time $O((|w| + |x|)d(w, x))$. Hence the total running time is $O(ND)$. □

4.3 Finding Good Matchings

In this section we consider the *Good Matching* problem, motivated in Section 4.1. We want to find an appropriate matching between the nodes of trees T_1 and T_2 that can serve as input to Algorithm *EditScript*.

As discussed in the introduction, if the data has object ids, then the matching problem is trivial. However, our focus here is on applications where information may not have keys or object-ids that can be used to match "fragments" of objects in one version with those in another. For example, the objects we are comparing, say sentences or paragraphs, may simply be characters with no meaningful object-id. In other cases the objects may have database identifiers but the ids may not be consistent between the two versions. For instance, the record representing a pillar in the architect's database may have id `778899`, but the same pillar in a subsequent version may have id `12345`. Here again, we need to match the pillars based on the value of the record (e.g., location and height of the pillar), as well as by its relationship to other objects (e.g., are the two pillars in the same room?). We use the term *keyless data* for hierarchical data that may not have identifying keys or object-ids. (Note that we are not ruling out keys for some objects; if they exist they can be used to match those objects quickly.)

When comparing versions of keyless data, there may be more than one way to match objects. Thus we need to define *matching criteria* that a matching must satisfy to be considered "good" or appropriate. In general, the matching criteria will depend on the domain being considered. One way of evaluating matchings that is desirable in many situations is to consider the minimum cost edit scripts that conform to the matchings (and transform T_1 into T_2). Intuitively, a matching that allows us to transform one tree to the other at a lower cost is a better matching. Formally, for matchings M and M', we say that *M is better than M'* if a minimum cost edit script that conforms to M is cheaper than a minimum cost edit script that conforms to M'. Our goal is to find a *best matching*, that is, a matching M that satisfies the given matching criteria and such that there is no better matching M' that also satisfies the criteria.

Unfortunately, if our matching criterion only requires that matched nodes have the same label, then finding the best matching has two difficulties. The first difficulty is that many matchings that satisfy only this trivial matching criterion may be unnatural in certain domains. For example, when matching documents, we may only want to match textual units (paragraphs, sections, subsections, etc.) that have more than a certain percentage of sentences in common. The second difficulty is one of complexity: the only algorithm known to us to compute the best matching as defined above (based on post-processing the output of the algorithm in [ZS89]) runs in time $O(n^2)$ where n is the number of tree nodes [Zha95]. To solve the first difficulty, we restrict the set of matchings we consider by introducing stronger matching criteria, as described below. These criteria also permit us to design efficient algorithms for matching. In the rest of this section, we describe some matching criteria for keyless data, using structured documents as an example.

4.3.1 Matching Criteria for Keyless Data

Our goal in this section is to augment the trivial label-matching criterion with additional criteria that simultaneously yield matchings that are meaningful in the domains of the data being considered, and that make possible efficient algorithms to compute a best matching.

Our first matching criterion states that nodes that are "too dissimilar" may not be matched with each other. For leaf nodes, this condition is stated as follows.

Matching Criterion 1 For leaf nodes $x \in T_1$ and $y \in T_2$, (x, y) can be in a matching only if $l(x) = l(y)$ and $compare(v(x), v(y)) \leq f$ for some parameter f such that $0 \leq f \leq 1$. (Recall that $l(x)$ and $v(x)$ denote the label and value of node x, and that *compare* is defined in Section 4.1.1 as a function used for determining the cost of updating a leaf node.) □

We also want to disallow matching internal nodes that do not have much in common. Here a more natural notion than the value (which is often null in the label-value model) is the number of common descendants. Let us say that an internal node x *contains* a node y if y is a leaf node descendent of x, and let $|x|$ denote the number of leaf nodes x contains. The following constraint allows internal nodes x and y to match only if at least a certain percentage of their leaves match (where t is a parameter):

Matching Criterion 2 Consider a matching M containing (x, y), where x is an internal node in T_1 and y is an internal node in T_2. Define

$$common(x, y) = \{(w, z) \in M \,|x \text{ contains } w, \text{and } y \text{ contains } z\}$$

Then in M we must have $l(x) = l(y)$ and

$$\frac{|common(x, y)|}{\max(|x|, |y|)} > t$$

for some t satisfying $\frac{1}{2} \leq t \leq 1$. □

Recall, from the introduction to this chapter, that one of the features of our work is that we use domain characteristics to design efficient algorithms. We now introduce these domain characteristics and formalize them by stating two assumptions that they let us make.

The hierarchical keyless data we are comparing has labels, and these labels usually follow a *structuring schema*, such as the one defined in [ACM95]. Many structuring schemas satisfy an *acyclic labels* condition, formalized in the following assumption:

Assumption 2 There is an ordering $<_l$ on the labels in the schema such that a node with label l_1 can appear as the descendent of a node with label l_2 only if $l_1 <_l l_2$.

In schemas where this condition is not satisfied, we can use domain semantics to merge labels that form a cycle, so that the resulting schema satisfies this condition.

Our next assumption states (informally) that the *compare* function is a good discriminator of leaves. It states that given any leaf s in the old document, there is at most one leaf in the new document that is "close" to s, and vice versa. For example, consider a world-wide web "movie database" source listing movies, actors, directors, etc. A tree representation of this data may contain movie titles as leaves. This assumption says that, when comparing two snapshots of this data, a movie title in one snapshot may "closely resemble" at most one movie title in the other.

Assumption 3 For any leaf $x \in T_1$, there is at most one leaf $y \in T_2$ such that $compare(v(x), v(y)) \leq 1$. Similarly, for any leaf $y \in T_2$, there is at most one leaf $x \in T_1$ such that $compare(v(x), v(y)) \leq 1$. □

This assumption may not hold for some domains. For example, legal documents may have many sentences that are almost identical. The algorithms we describe below are guaranteed to produce an optimal matching when Assumption 3 holds. When Assumption 3 does not hold, our algorithm may generate a suboptimal, but still correct, solution. However, we can often post-process such a suboptimal solution to obtain an optimal solution. We discuss this issue further in Section 4.6.

Matching Criteria 1 and 2 and the assumptions that we have introduced above allow us to simplify the best matching problem as follows. (Recall that a best matching is a matching that can be used to produce an edit script of the lowest cost among all matchings satisfying the Matching Criteria.) We say that a matching is *maximal* if it is not possible to augment it without violating the Matching Criteria. We can show that our Matching Criteria imply that there is a unique maximal matching. Furthermore, given our assumptions, we can show that this unique maximal matching is also the best matching. These statements are formalized in Theorem 4 stated below after a couple of preliminary lemmas.

Lemma 2 *For matchings M and M' that satisfy Matching Criterion 1 if $M \subseteq M'$ then M is not better than M'.* $\square$

Proof. For matchings M and M' satisfying the value constraint, the cost of moving and then updating a node is no more than the cost of deleting and inserting a node. Suppose M' is obtained from M by adding to M the match (x, y). Then any edit script conforming to M will contain operations that delete the node x and insert another node corresponding to y, whereas an edit script conforming to M' can replace the insertion and deletion by a move and an update and be no more expensive. $\square$

Lemma 3 *Suppose T_1 and T_2 satisfy the acyclicity condition for labels and Assumption 3 holds. For any internal node $x \in T_1$, there is at most one internal node $y \in T_2$ such that the pair (x, y) satisfies the match threshold constraint. Similarly, for any internal node $y \in T_2$, there is at most one internal node $x \in T_1$ such that the pair (x, y) satisfies the match threshold constraint.* $\square$

Proof. Suppose that node $x \in T_1$ has two "partners" y and z in T_2 satisfying the match threshold constraint. Then we must have

$$\frac{|common(x, y)|}{\max(|x|, |y|)} > t$$

```
1. $M \leftarrow \phi$
2. Mark all nodes of $T_1$ and $T_2$ "unmatched."
3. Proceed bottom-up on tree $T_1$
       For each unmatched node $x \in T_1$, if there is an unmatched node $y \in T_2$ such that
       $equal(x, y)$ then
         i. Add $(x, y)$ to $M$.
        ii. Mark $x$ and $y$ "matched."
```

Figure 4.10: Algorithm *Match*

and

$$\frac{|common(x,z)|}{\max(|x|,|z|)} > t.$$

The acyclicity condition implies that y and z can have no common descendents, so we must have

$$|common(x,y)| + |common(x,z)| > 2t|x|$$

which is impossible since $t \geq 1/2$. A symmetric argument holds, reversing T_1 and T_2.
□

Theorem 4 (Unique Maximal Matching) If T_1 and T_2 are trees satisfying Matching Criteria 1 and 2 and Assumptions 2 and 3, then there is a unique maximal matching M of the nodes of T_1 and T_2. Moreover, M is also the unique best matching that satisfies the matching criteria. □

Proof. Follows directly from Lemmas 2 and 3. □

4.3.2 A Simple Matching Algorithm

Theorem 4 allows us to construct a straightforward algorithm to obtain the best matching that satisfies our matching criteria. For each node $x \in T_1$, we simply compare x with each unmatched node $y \in T_2$ that has the same label as x. We use the following function *equal* for leaf nodes, where f is a parameter such that

$0 \le f \le 1$:

$$equal(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and } compare(v(x), v(y)) \le f \\ false & \text{otherwise} \end{cases}$$

We use the following function *equal* for internal nodes ($t > \frac{1}{2}$ is a parameter):

$$equal(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and } \frac{|common(x,y)|}{\max(|x|,|y|)} > t \\ false & \text{otherwise} \end{cases}$$

The algorithm must match leaves before matching internal nodes to ensure that the equality function for internal nodes can be evaluated. Figure 4.10 shows this simple matching algorithm, which we call Algorithm *Match*.

Example 4.3.1 We illustrate our simple matching algorithm on the trees from our running example in Figure 4.1. The algorithm first examines each leaf node of T_1 in turn, and attempts to pair it with a leaf node of T_2. This process produces the following matching of leaf nodes:

$$M = \{(5, 15), (7, 16), (8, 18), (9, 19), (10, 17)\}$$

The algorithm then tries to pair nodes with the label `P`, and adds the pairs $(2, 12)$, $(3, 14)$, and $(4, 13)$ to the matching. Finally, pairing nodes with label `D` yields the pair $(1, 11)$. The final matching that results is shown in Figure 4.1 using dashed lines. □

In Section 4.3.4 we show that the running time of Algorithm *Match* is proportional to

$$n^2c + mn \tag{4.1}$$

where n is the total number of leaf nodes in T_1 and T_2, m is the total number of internal nodes in T_1 and T_2, and c is the average cost of executing $compare(x, y)$ for a pair of leaf nodes x and y. (Section 4.5 describes how we compare sentences in our implementation.)

```
1. $M \leftarrow \phi$
2. For each leaf label $l$ do
   (a) $S_1 \leftarrow chain_{T_1}(l)$.
   (b) $S_2 \leftarrow chain_{T_2}(l)$.
   (c) $lcs \leftarrow LCS(S_1, S_2, equal)$.
   (d) For each pair of nodes $(x, y) \in lcs$, add $(x, y)$ to $M$.
   (e) Pair unmatched nodes with label $l$ as in Algorithm Match, adding matches to $M$.
3. Repeat steps 2a through 2e for each internal node label $l$.
```

Figure 4.11: Algorithm *FastMatch*

4.3.3 A Faster Matching Algorithm

We can significantly reduce the number of comparisons in Algorithm *Match* when T_1 and T_2 are nearly alike, which is often the case in practice. We modify Algorithm *Match* to Algorithm *FastMatch*, shown in Figure 4.11. Algorithm *FastMatch* uses the longest common subsequence (LCS) routine, introduced earlier in Section 4.2.2, to perform an initial matching of nodes that appear in the same order. Nodes still unmatched after the call to LCS are processed as in Algorithm *Match*. The function *equal* in the LCS call is as defined in Section 4.3.2.

In Algorithm *FastMatch* we assume that all nodes with a given label l in tree T are chained together from left to right. Let $chain_T(l)$ denote the chain of nodes with label l in tree T. Node x occurs to the left of node y in $chain_T(l)$ if x appears before y in the in-order traversal of T when siblings are visited left-to-right.

To help us analyze the running time of Algorithm *FastMatch*, we define the *weighted edit distance* e between trees T_1 and T_2 as follows. Let $E = e_1 e_2 \ldots e_n$ be the shortest edit script that transforms T_1 to T_2. Then the weighted edit distance is given by

$$e = \sum_{1 \leq i \leq n} w_i$$

where w_i, for $1 \leq i \leq n$, is defined as follows:

$$w_i = \begin{cases} 1 & \text{if } e_i \text{ is an insert or a delete} \\ |x| & \text{if } e_i \text{ is a move of the subtree rooted at node } x \\ 0 & \text{if } e_i \text{ is an update} \end{cases}$$

Recall that $|x|$ denotes the number of leaf nodes that are descendants of node x. Intuitively, the weighted edit distance indicates how structurally different the two trees are, where the degree of difference associated with moving a subtree is given by the number of leaves in that subtree.

In Section 4.3.4 below we show that the running time of Algorithm *FastMatch* is proportional to

$$(ne + e^2)c + 2lne \tag{4.2}$$

where n and c are the same as in Formula (4.1) of Section 4.3.2, l is the number of internal node labels, and e is the weighted edit distance between T_1 and T_2. A comparison of Formula (4.2) with Formula (4.1) shows that Algorithm *FastMatch* is substantially faster than Algorithm *Match* when e is small compared to n, as is typically the case. Section 4.6 presents results from our empirical performance study of Algorithm *FastMatch*.

4.3.4 Analysis of Matching Algorithms

For a label a, let n_a be the total number of nodes with label a in T_1 and T_2. Let c_a be the average cost of computing $equal(x, y)$ for nodes x and y with label a. Then Algorithm *Match* takes time $O(n_a^2 c_a)$ to match nodes with label a. Thus, the total time taken by the algorithm is proportional to $\sum_{a \in L} n_a^2 c_a$, where L is the set of all labels that appear in T_1 or T_2.

To simplify our analysis, let us assume that L is made up two disjoint subsets of labels—P, the set of labels of leaf nodes, and Q, the set of labels of internal nodes. Further, let us assume that all leaf node comparisons have the same average cost, that is, $c_a = c$ for all $a \in P$. Let n be the total number of leaf nodes in T_1 and T_2. Then matching leaf nodes takes time $O(n^2 c)$. For an internal node label $b \in Q$, computing *equal*(x, y) for nodes x and y with label b requires us to intersect the leaf nodes they contain, which takes time proportional to $\min(|x|, |y|)$. If we assume that, on average, $|x| = n/n_b$ for nodes x with label b, then we may approximate c_b by n/n_b, and so matching nodes with label b takes time $O(n_b n)$. Thus, the total time taken by Algorithm *Match* is proportional to

$$n^2 c + n \sum_{b \in Q} n_b.$$

If we denote by m the total number of internal nodes in T_1 and T_2, then $m = \sum_{b \in Q} n_b$, and so the running time of Algorithm *Match* is $O(n^2 c + mn)$.

For a label a, let $d_a = n_a - lcs_a$. Then Algorithm *FastMatch* takes time proportional to $(n_a d_a + d_a^2) c_a$ to match nodes with label a. Let us make the same assumptions as in the analysis of Algorithm *Match*. Then matching leaf nodes takes time that is proportional to $(nd + d^2)c$, where $d = \sum_{a \in P} d_a$. For internal nodes with label b, let us once again assume that $c_b = n/n_b$. Now, remembering that $d_b \leq n_b$ and so $d_b^2 c_b \leq n d_b$, the time taken to match nodes with label b is proportional to $2nd_b$. Hence the total time taken by Algorithm *FastMatch* is proportional to

$$(nd + d^2)c + \sum_{b \in Q} 2nd_b.$$

Now let e be the weighted edit distance between trees T_1 and T_2, as defined in Section 4.3.3. It is clear that for any label b, we have $d_b \leq e$. Hence the running time of Algorithm *FastMatch* is bounded by

$$(ne + e^2)c + 2lne$$

where $l = |Q|$ is the number of labels of internal nodes in T_1 and T_2.

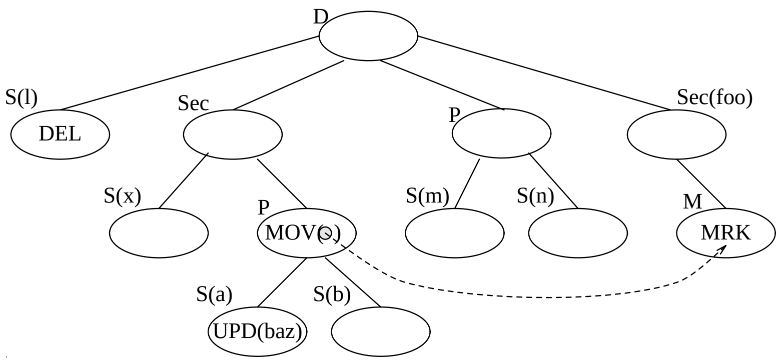


Figure 4.12: Delta tree for edit script in Example 4.1.1

4.4 Delta Trees

In this section we describe a representation for deltas in hierarchically structured data that is more natural and useful than edit scripts for certain scenarios. As we have seen above, an edit script gives us the sequence of operations needed to transform one tree to another, and thus is a simple "operational" representation of deltas. One problem with edit scripts is that they refer to tree nodes using node identifiers. Node identifiers may be system-generated and thus not meaningful to the user. Furthermore, the flat, sequential structure of an edit script may make it difficult to use for querying and browsing hierarchical deltas.

In a relational database, deltas usually are represented using *delta relations*: For a relation R, delta relations *inserted(R)* and *deleted(R)* contain the tuples inserted to and deleted from R, while delta relations *old-updated(R)* and *new-updated(R)* contain the old and new values of updated tuples [GHJ$^+$93, WC96b]. One can contrast this representation with the relational version of an edit script, which would (presumably) be a list of tuple-level inserts, deletes, and updates, possibly based on tuple identifiers. We are interested in a representation comparable to delta relations but for hierarchically structured data.

We define a structure called a *delta tree* for representing deltas. Intuitively, one can think of a delta tree as "overlaying" an edit script onto the data using node annotations. (In this sense, a delta tree differs from a delta relation in that delta relations are kept separate from the original data. In practice delta relations often are joined with their corresponding relation [WC96b], and we are effectively representing this join explicitly.) As an example, the delta tree corresponding to the edit script from Example 4.1.1 is shown in Figure 4.12. Note that we do not need node identifiers since the annotated nodes are at the appropriate positions in the delta tree.

More formally, let T_1 and T_2 be two trees. A delta tree for T_1 with respect T_2 is a tree ΔT such that, in addition to a label and value, each node in ΔT has exactly one of the following *annotations*:

- IDN, indicating that the node corresponds to a node in the original tree. (In Figure 4.12, IDN annotations appear as blanks.)
- UPD(v), indicating that the value of the node is updated to v.
- INS(l,v), indicating that the node is inserted with label l and value v.
- DEL, indicating deletion of the subtree rooted at the node.
- MOV(x), indicating that the node is moved to the position of the "marker node" x.
- MRK, indicating that the node is the destination of a move operation.

A *correct* delta tree for T_1 with respect to T_2 must have the property that there is at least one edit script E such that:

1. E transforms T_1 to T_2.
2. There is a total order over the nodes of ΔT such that outputting the edit operations corresponding to the node annotations in this order yields edit script E.

Note that there may be more than one such edit script. In general, we are interested in correct delta trees corresponding to minimum cost edit scripts.

In our implementation of the algorithms described in Sections 4.2 and 4.3, we construct the delta tree directly as a side-effect of producing an edit script. Essentially, this is achieved by modifying algorithm *EditScript* (Section 4.2) to emit a call to add a node to the delta tree every time an operation is added to the edit script being computed. Our implementation uses the delta tree representation rather than the edit script in order to produce meaningful output, as described in the next section.

4.5 Implementation

To validate our method for computing and representing deltas, as well as to have a vehicle for studying the performance of our algorithms, we have implemented a program for computing and representing changes in structured documents. Below, we describe the implementation of this program, called *LaDiff*. We focus on Latex documents, but the implementation can easily handle other kinds of structured documents (e.g., HTML) by changing the parsing routines. Our performance study is presented in Section 4.6.

LaDiff takes as input two files containing the old and new versions of a Latex document. These files are first parsed to produce their tree representations (the old tree and new tree, respectively). Currently, we parse a subset of Latex consisting of sentences, paragraphs, subsections, sections, lists, items, and document. It is easy to extend our parser to handle a larger subset of Latex , and we plan to do so in the future. Next, the edit script and delta tree are computed using the algorithms of Sections 4.2–4.3. Our program takes the match threshold t (Section 4.3) as a parameter. Our comparison function for leaf nodes—which are sentences—first computes the LCS (recall Section 4.2.2) of the words in the sentences, then counts the number of words not in the LCS. Interior nodes (paragraphs, items, sections, etc.) are compared as described in Section 4.3. Finally, a preorder traversal of the delta tree is performed to produce an output Latex document with annotations describing the changes.

We now illustrate a sample run of *LaDiff*. We show only a short example, based on an excerpt from the TEXbook [Knu86], that illustrates some of the change detection features. Our implementation uses a modified version of the LCS algorithm from [Mye86]. Note that we cannot use the LCS algorithm used by the standard UNIX *diff* program, because it requires inequality comparisons in addition to equality comparisons.

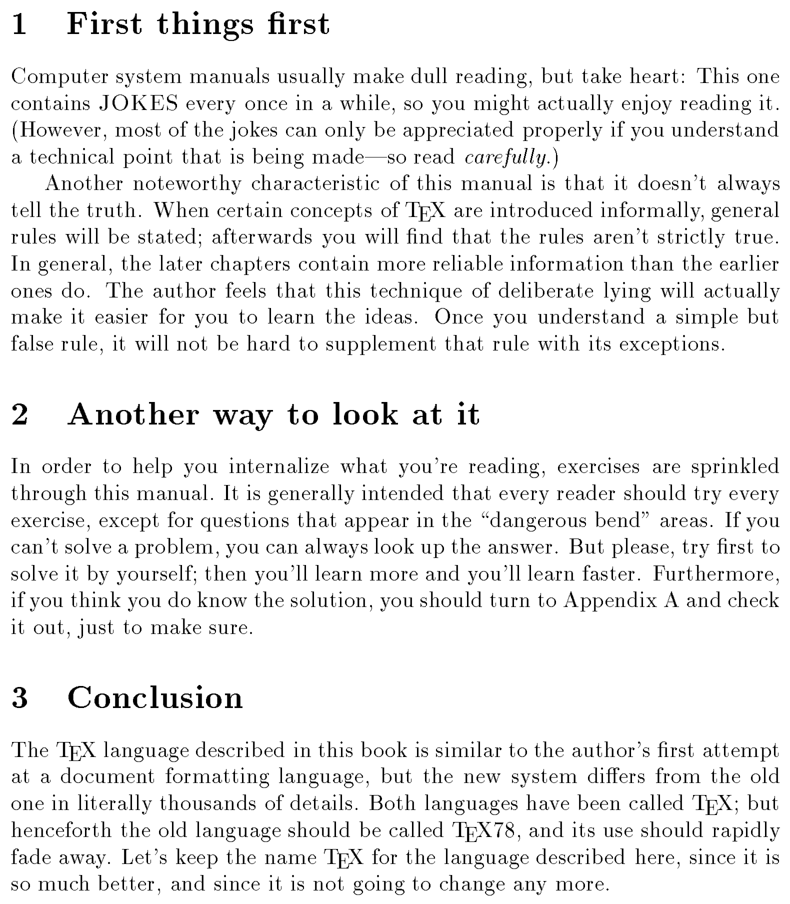
Figures 4.13 and 4.14 show the old and new versions of the example document. We tried the UNIX *diff* program on these documents, and the output was not very useful. Figure 4.15 shows the output of *LaDiff*. The conventions used by *LaDiff* for marking various changes in the output document are shown in Table 4.1. Sentence level changes are marked using changes in font: inserted sentences are in bold font, while deleted and updated sentences are in small and italic fonts respectively. Sentence moves are marked by putting the sentence in small font, labeling it, and referencing the label with a footnote at the new position of the sentence. (See the first and last sentences in the third section in Figure 4.15, for example.) Paragraph changes are marked using marginal notes indicating whether the paragraph is inserted, deleted, moved, or updated. In the case of paragraph moves, the old position of the paragraph is marked with a label which is referenced from the marginal note in its new position. (See the third paragraph in Figure 4.15, for example.) Changes in sections, subsections, and itemized lists are marked using similar schemes, as summarized by the table.

Note that sentences, as well as other textual units, may be moved and updated at the same time. The mark-up conventions used by *LaDiff* allow us to mark these changes simultaneously. For example, the first sentence in Figure 4.15 is in italic font, indicating that it was updated, and also has a footnote telling us that it was moved from position S1 (near the end of the document).

We can see that *LaDiff* properly detects insertions, deletions, updates, and moves of sentences and paragraphs. Representing the changes in an intuitive manner is a challenging problem, and we plan to work on it further.

4.6 Empirical evaluation of FastMatch

In Section 4.3 we presented Algorithm *FastMatch* to find a matching between two trees, and we stated that its running time is given by an expression of the form r_1c+r_2. In this expression, r_1 represents the number of leaf node comparisons (invocations of



1 First things first

Computer system manuals usually make dull reading, but take heart: This one contains JOKES every once in a while, so you might actually enjoy reading it. (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully.*)

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. When certain concepts of TEX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

2 Another way to look at it

In order to help you internalize what you're reading, exercises are sprinkled through this manual. It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure.

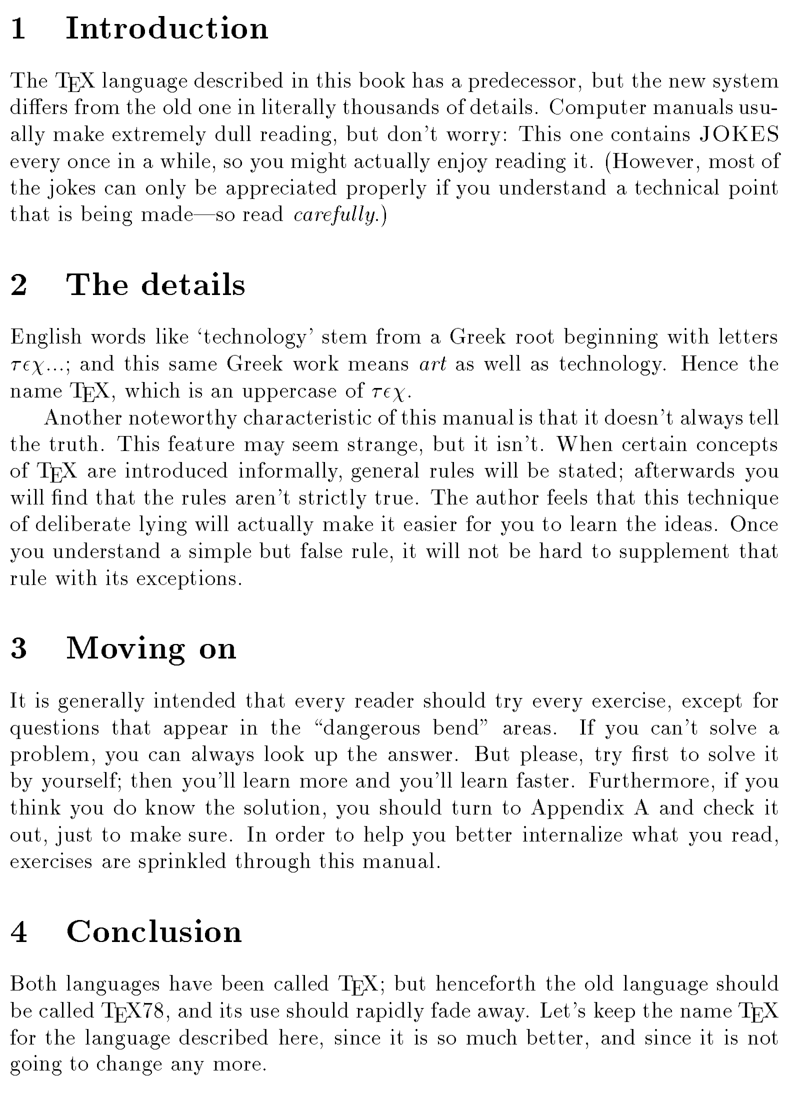
3 Conclusion

The TEX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details. Both languages have been called TEX; but henceforth the old language should be called TEX78, and its use should rapidly fade away. Let's keep the name TEX for the language described here, since it is so much better, and since it is not going to change any more.

Figure 4.13: Old version of document

Textual Unit	Edit Operation			
	Insert	Delete	Update	Move
Sentence	Bold font	Small font	Italic font	Footnote, label
Paragraph	Marginal note			Marginal note, label
Item	Marginal note			Marginal note, label
Subsection	Annotation(ins,del,upd,mov) in heading			
Section	Annotation(ins,del,upd,mov) in heading			

Table 4.1: Mark-up conventions used by *LaDiff*.



1 Introduction

The TeX language described in this book has a predecessor, but the new system differs from the old one in literally thousands of details. Computer manuals usually make extremely dull reading, but don't worry: This one contains JOKES every once in a while, so you might actually enjoy reading it. (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully*.)

2 The details

English words like 'technology' stem from a Greek root beginning with letters $\tau\epsilon\chi$...; and this same Greek work means *art* as well as technology. Hence the name TeX, which is an uppercase of $\tau\epsilon\chi$.

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. This feature may seem strange, but it isn't. When certain concepts of TeX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

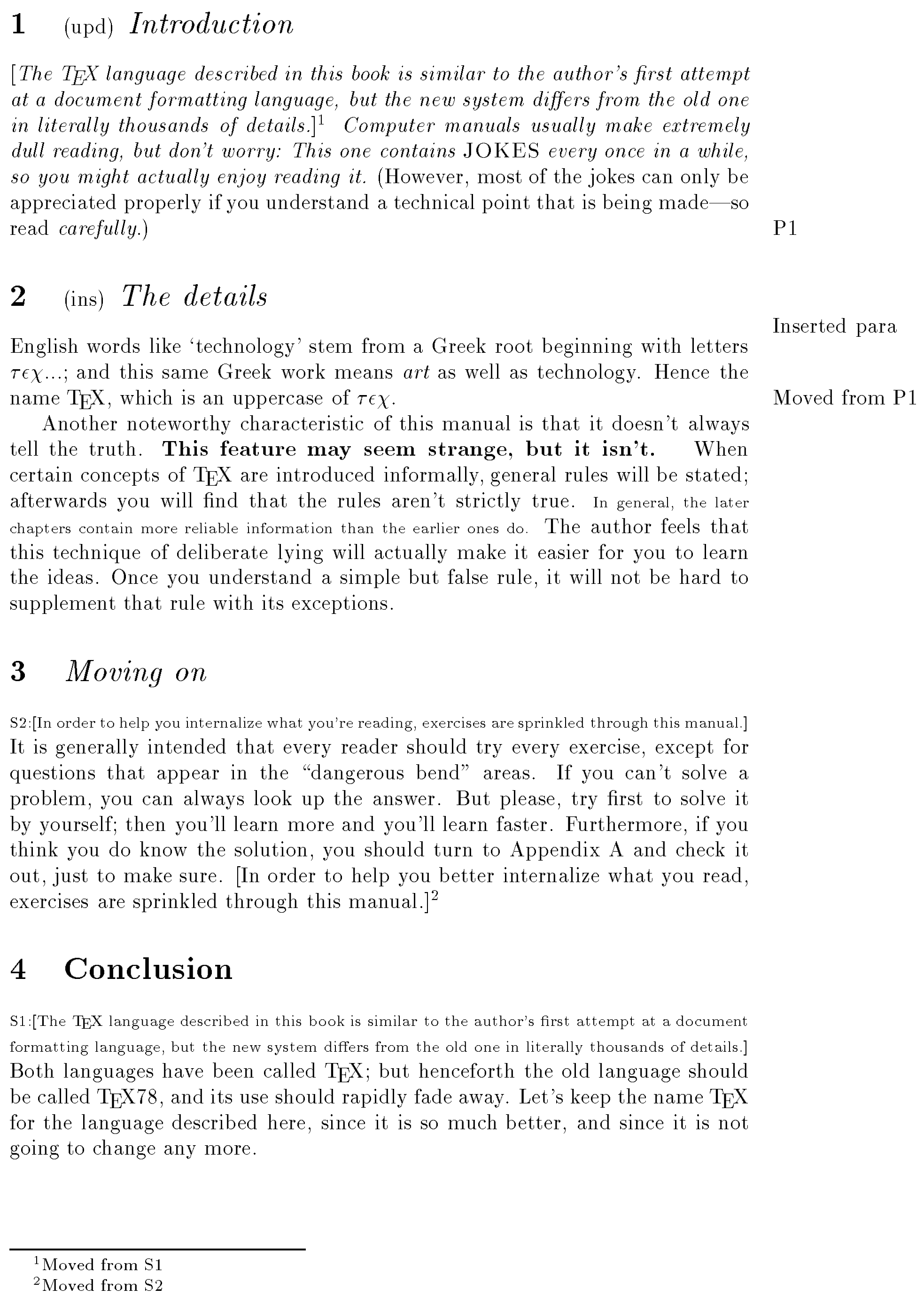
3 Moving on

It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure. In order to help you better internalize what you read, exercises are sprinkled through this manual.

4 Conclusion

Both languages have been called TeX; but henceforth the old language should be called TeX78, and its use should rapidly fade away. Let's keep the name TeX for the language described here, since it is so much better, and since it is not going to change any more.

Figure 4.14: New version of document



1 (upd) *Introduction*

[*The TEX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details.*][1] *Computer manuals usually make extremely dull reading, but don't worry: This one contains* JOKES *every once in a while, so you might actually enjoy reading it.* (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully.*) P1

2 (ins) *The details*

Inserted para

English words like 'technology' stem from a Greek root beginning with letters $\tau\epsilon\chi$...; and this same Greek work means *art* as well as technology. Hence the name TEX, which is an uppercase of $\tau\epsilon\chi$. Moved from P1

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. **This feature may seem strange, but it isn't.** When certain concepts of TEX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

3 *Moving on*

S2:[In order to help you internalize what you're reading, exercises are sprinkled through this manual.] It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure. [In order to help you better internalize what you read, exercises are sprinkled through this manual.][2]

4 **Conclusion**

S1:[The TEX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details.] Both languages have been called TEX; but henceforth the old language should be called TEX78, and its use should rapidly fade away. Let's keep the name TEX for the language described here, since it is so much better, and since it is not going to change any more.

[1]Moved from S1
[2]Moved from S2

Figure 4.15: Output document (marked up)

function *compare*), c is the average cost of comparing leaf nodes, and r_2 represents the number of node partner checks. Partner checks are implemented in *LaDiff* as integer comparisons. We know that r_1 is bounded by $(ne + e^2)$, and that r_2 is bounded by $2lne$, where n is the number of tree nodes, e is the weighted edit distance between the two trees, and l is the number of internal node labels. The parameter e depends on the nature of the differences between the trees (recall the definition of weighted edit distance in Section 4.3.3).

There are two reasons for studying the performance of FastMatch empirically. The first reason is that the formula for the running time contains the weighted edit distance, e, which is difficult to estimate in terms of the input. A more natural measure of the input size is the number of edit operations in an optimal edit script, which we call the *unweighted* edit distance, d. We can show analytically that the ratio e/d is bounded by $\log n$ for a large class of inputs, but we believe that in real cases, its value is much lower than $\log n$. We therefore study the relationship between e and d empirically. The second reason is that we would like to test our conjecture that the analytical bound on the running time of FastMatch is "loose," and in most practical situations the algorithm runs much faster.

For our performance study, we used three sets of files. The files in each set represent different versions of a document (a conference paper). We ran FastMatch on pairs of files within each of these three sets. (Comparing files across sets is not meaningful because we would be comparing two completely different documents.) In Figure 4.16 we indicate how the weighted edit distance (e) varies with the unweighted edit distance (d), for each of the three document sets. Recall that n is the number of tree leaves, that is, the number of sentences in the document. We see that the relationship between e and d is close to linear. Furthermore, the variance with respect to the three document sets is not high, suggesting that e/d is not very sensitive to the size of the documents (n). The average value of e/d is 3.4 for these documents.

In Figure 4.17 we plot how the running time of FastMatch varies with the weighted edit distance e. The vertical axis is the running time as measured by the number of comparisons made by FastMatch and the horizontal axis is the weighted edit distance. Note that the analytical bound on the number of comparisons made by FastMatch

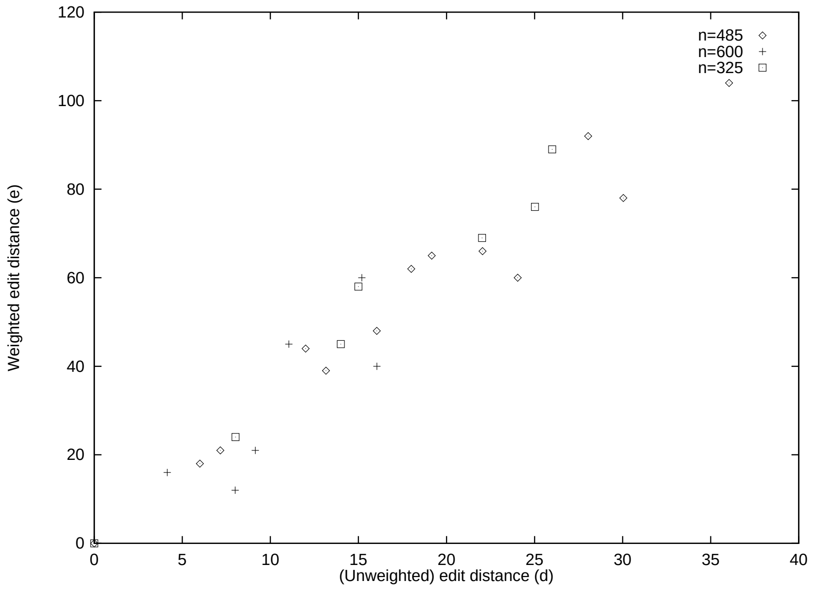


Figure 4.16: Relation between the weighted and unweighted edit distances

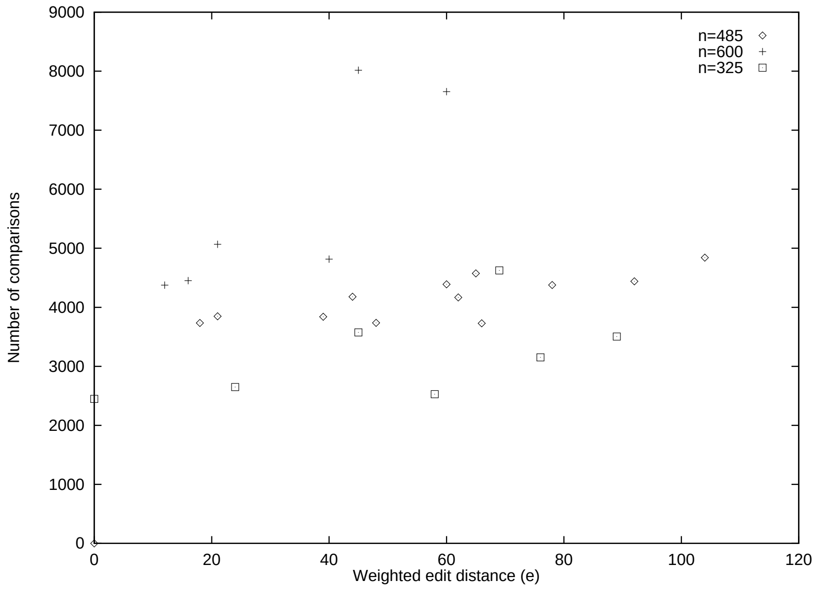


Figure 4.17: Running time of FastMatch

is much higher than the numbers depicted in Figure 4.17; on the average, FastMatch makes approximately 20 times fewer comparisons than those predicted by the analytical bound, supporting our conjecture that the analytical bound on the running time is a loose one. We also observe that Figure 4.17 suggests an approximately linear relation between the running time and e, although there is a high variance. This variance may be explained by our first observation that the actual running time is far below the predicted bound.

Another issue that needs to be addressed is the effect of Assumption 3 on the quality of the solution produced by FastMatch. Recall from Section 4.3 that FastMatch is guaranteed to produce an optimal matching only when Assumption 3 holds. When Assumption 3 does not hold, the algorithm may produce a suboptimal matching. We describe a post-processing step that, when added to FastMatch, enables us to convert the possibly suboptimal matching produced by FastMatch into an optimal one in many cases: Proceeding top-down, we consider each tree node x in turn. Let y be the partner of x according to the current matching. For each child c of x that is matched to a node c' such that $parent(c') \neq y$, we check if we can match c to a child c'' of y, such that $compare(c, c'') \leq f$, where f is the parameter used in Matching Criterion 1. If so, we change the current matching to make c match c''. This post-processing phase removes some of the suboptimalities that may be introduced if Assumption 3 does not hold for all nodes.

Even with post-processing, it is still possible to have a suboptimal solution, as follows: Recall that FastMatch begins by matching leaves, and then proceeds to match higher levels in the tree in a bottom-up manner. With this approach, a mismatch at a lower level may "propagate," causing a mismatch at one or more higher levels. Our post-processing step will correct all mismatches other than those that propagated from lower levels to higher levels. It is difficult to evaluate precisely those cases that in which this propagation occurs without performing exhaustive computations. However, we can derive a necessary (but not sufficient) condition for propagation, and then measure that condition in our experiments. Informally, this condition states that in order to be mismatched, a node must have more than a certain number of children that violate Assumption 3, where the exact number depends on the match threshold

Match threshold (t):	0.5	0.6	0.7	0.8	0.9	1.0
Upper bound on mismatches (%):	0.4	1	3	7	9	10

Table 4.2: Mismatched paragraphs in *FastMatch*.

t. Actually, this condition is weak; a node must satisfy many other conditions for the possibility of a mismatch to exist, and even then a mismatch is not guaranteed.

For the same document data analyzed earlier, Table 4.2 shows some statistics on the percentage of paragraphs that *may* be mismatched for a given value of the match threshold t. For example, we see that with $t = 0.6$, we may mismatch at most 1% of the paragraphs. A lower value of t results in a lower number of possible mismatches. We see that the number of mismatched paragraphs is low, supporting our claim. Since the condition used to determine when a mismatch may occur is a weak one, the percentage of mismatches is expected to be much lower than suggested by these numbers. Furthermore, note that a suboptimal matching compromises only the quality of an edit script produced as the final output, not its correctness. In many applications, this trade-off between optimality and efficiency is a reasonable one. For example, when computing the delta between two documents, it is often not critical if the edit script produced is slightly longer than the optimal one.

4.7 Summary

In this chapter, we studied the problem of detecting changes from snapshots of structured or semistructured data that is represented using ordered trees. We formalized the change detection problem as the problem of finding a minimum-cost edit script that transforms one given tree to the other. We defined an edit script to be a sequence of operations that may insert or delete a node, update the label of a node, or move a subtree. We described the benefits of modeling changes using not only the traditional insert, delete, and update operations, but also the powerful subtree move operation that we introduced in this chapter.

Our solution to this change detection problem is based on the use of a matching between the nodes of the two input trees. The relation between matchings and edit scripts is formalized by our definition of the conformance of an edit script to a matching. Using this definition of conformance, we described a two-step strategy to solve the change detection problem. We first described the second step: Given a matching between the two input trees, we presented a method for computing a minimum-cost edit script that conforms to that matching. Next, we presented methods for computing such a matching in the first place. We proved that our methods result in an optimal solution under some reasonable assumptions.

We studied our algorithms both analytically and empirically. By making use of domain characteristics, our algorithms are able to compute differences significantly faster than those studied in prior work. We proved that an upper bound on the number of comparisons made by our FastMatch algorithm is $(ne + e^2)c + 2lne$, where n is the number of tree nodes, e is the weighted edit distance between the trees, l is the number of interior node labels, and c is the cost of the function used to compare values of leaf nodes. We showed empirically that e is typically a small constant times the unweighted edit distance. Further, we showed empirically that for the dataset studied, the number of comparisons made by FastMatch are approximately 20 times smaller than the analytical bound. Although the results of our methods are guaranteed to always be correct, they are guaranteed to be optimal only when the values in the leaf nodes of the input trees are not too similar to each other. We described a simple postprocessing step that results in optimal solutions for a large class of inputs that do not satisfy this assumption.

While edit scripts provide a good theoretical basis for computing the differences between two trees, they are not convenient for storing and browsing such differences. We therefore defined delta trees, which store differences alongside the data they modify, and thus allow us to conveniently browse data marked up with the detected changes. In Chapter 7, we describe how the idea of delta trees is extended to provide a general purpose data model and query language for changes in semistructured data. Finally, we illustrated the application of these ideas by describing our implementation of a program for computing and presenting changes in structured documents.

In the next two chapters, we continue our study of the change detection problem.

In Chapter 5, we address the problem of detecting changes when data is represented using unordered trees that do not have the layered structure assumed in this chapter. Further, we allow subtrees to be not only moved, but also copied and uncopied. By using the subtree operations move, copy, and uncopy, edit scripts can describe changes in a succinct and intuitively appealing manner. For example, when comparing two versions of a license agreement, we can detect not only sentences that have been moved from one paragraph to another, but also sentences and paragraphs that have been copied. However, these subtree operations may also be combined in a complicated manner resulting in edit scripts that are intuitively unusable. For example, an edit script may repeatedly copy, move, and uncopy slightly different portions of a subtree, resulting in a change description that is very difficult to understand and use. In Chapter 5, we discuss such complications in detail and present a solution based on restricting edit script to disallow problematic sequences of edit operations.

In Chapter 6 we explore an alternative formulation of the change detection problem. Instead of the conventional linear edit script model, we use a model of tree transformations that is based on the idea of applying edit operations in parallel. This formulation allows us overcome the difficulties due to problematic sequences of edit operations in a manner that is simpler and more elegant than the solution based on linear edit scripts. This model of parallel transformations also results in simpler algorithms for change detection.

Chapter 5

Detecting Changes in Unordered Trees

In Chapter 4, we presented algorithms for comparing snapshots of data that is represented using ordered trees. Ordered trees are a natural abstraction of structured or semistructured data that has a meaningful order among components. For example, documents consist of sections that have ordered paragraphs as components, paragraphs consist of an ordered list of sentences, and so on. However, we often encounter data that has no inherent ordering. For example, consider the set of students in a class, or the result of a query asking for stores selling a certain product. We need to model such data using unordered trees.

In this chapter, we present techniques to compare unordered trees by computing a minimum-cost edit script that transforms one tree to the other. In addition to modeling unordered trees, in this chapter we also extend our set of tree edit operations by adding operations to copy and uncopy subtrees. Similar to the subtree move operations described in Chapter 4, these new subtree operations result in more meaningful and usable results when comparing data. Further, unlike in Chapter 4 where we imposed significant restrictions on our edit scripts to yield a more efficient algorithm, in this chapter we follow a more general approach that is applicable to a larger collection of data. Instead of assuming that the data has certain characteristics, such as "few" duplicates, we present algorithm MH-DIFF (for meaningful, hierarchical difference), which handles all data, with performance improving when the input data has certain characteristics.

5.1 Introduction

We describe tree differences using *move*, *copy*, and *uncopy* operations in addition to the more traditional insert, delete, and update operations. Thus, if a substructure (e.g., a section of text, a shift register) is moved to another location, our algorithm will report it as a single operation. (This feature is shared by our ordered tree comparison algorithm described in Chapter 4.) Traditional change detection algorithms report such changes using an edit script that deletes all the nodes in the moved subtree and then inserts identical nodes at the new location of the subtree. An application or person using such an edit script is unable to easily detect that the deleted and inserted nodes are simply components of a subtree move operation. Similarly, if the substructure is copied (e.g., a second shift register is added which is identical to one already in the circuit), then our algorithm will identify it as such. Traditional change detection algorithms (and our algorithm in Chapter 4) report such changes using a sequence of node insertion operations, thus losing the information that the new nodes are simply copies of some existing nodes.

Note that detecting moves and copies becomes more important if the moved or copied subtree is large. For instance, if we are comparing file systems, and a large directory with thousands of files is mounted elsewhere, we clearly do not wish to report the change as thousands of file deletes followed by thousands of file creations.

The problem of comparing unordered trees is inherently more complex than the analogous problem for ordered trees. Most formulations of this problem (including ours, described in this chapter) are NP-hard. Even a simple problem formulation that uses only insert, delete, and update operations can be shown to be NP-hard by reduction from the "exact cover by three-sets" problem [ZWS95].

Algorithm MH-DIFF provides a heuristic solution, which is based on transforming the problem to the "edge cover domain." That is, instead of working with edit scripts, the algorithm works with edge covers that represent how one set of nodes match another set. In this transformation, the costs of the edit operations are translated into costs on the edges of the cover.

In Chapter 4 we defined a variant of the change detection problem for ordered trees, using subtree *moves* as an edit operation in addition to insertions, deletions, and updates, and presented an efficient algorithm for solving it. That algorithm uses domain characteristics to find a solution efficiently. A major drawback of the algorithm in Chapter 4 is that it assumes that the number of duplicates (or near duplicates) in the labels found in the input trees is very small. Another drawback of of the algorithm in Chapter 4 is that it assumes each node of the input trees has a special tag that describes its semantics. (For example, an ordered tree representing a document may have tags "paragraph," "section," etc.) Furthermore, that algorithm assumes the existence of a total order $<_t$ over these tags such that a node with tag t_1 cannot be the child of a node with tag t_2 unless $t_1 \leq t_2$. While these assumptions are reasonable in a text comparison scenario, there are many domains in which they do not hold. Here, on the other hand, here we drop these assumptions, and introduce copy operations. This leads to an algorithm that is very different from the one in Chapter 4, and that yields a heuristic solution in worst-case $O(n^3)$ time, where n is the number of nodes, but most often in roughly $O(n^2)$ time.

In summary, the main contributions of this chapter are the following:

- We formulate a change detection problem for unordered trees. Our formulation includes move and copy operations, and uses a flexible cost model for edit operations.

- We present MH-DIFF, an efficient algorithm for computing meaningful edit-scripts that are very close to the minimal cost edit script.

- We present experimental results showing how close to optimal the MH-DIFF solutions are. We also experimentally evaluate the key parameter that determines the running time of MH-DIFF in practice.

The rest of this chapter is organized as follows. In Section 5.2, we describe the data model used in this chapter and present the formal definition of the change detection problem that we study in this chapter. Section 5.3 presents a quick overview of our algorithm for solving this problem. In Section 5.4, we describe how the essential features of an edit script are represented using an edge cover. We define the edge cover representing an edit script, and present an algorithm for recovering an edit script from such an edge cover. This correspondence between edit scripts and edge covers allows us to compute a minimum-cost edit script by first finding the corresponding edge cover. Section 5.5 describes how such an edge cover is found. In Section 5.6, we present our implementation of MH-DIFF and briefly describe its performance. A detailed performance study is presented in Chapter 9. We summarize the chapter in Section 5.7.

5.2 Model and Problem Definition

We use rooted, labeled trees as our model for structured data. These are trees in which each node n has a label $l(n)$ that is chosen from an arbitrary domain $\mathcal{L}$. Unlike the ordered trees studied in Chapter 4, these trees do not specify an order among the children of a node. Unordered trees are a natural abstraction of several kinds of data in the *Object Exchange Model (OEM)* (introduced in Chapter 3). The type of tree (ordered or unordered) best suited to represent some OEM data depends on the nature of the data and its domain. For example, if we are comparing OEM representations of structured documents, which have an inherent order among their components, an abstraction using ordered trees is natural. On the other hand, if we are comparing OEM representations of semistructured databases (introduced in Chapter 1) describing books in a library, an abstraction using unordered trees is natural.

As in Chapter 4, the problem of snapshot change detection in structured data is thus the problem of finding a way to edit the tree representation of one snapshot to that of the other. (However, as described below, in this chapter we use a larger set of edit operations to describe changes.) We denote a tree T by its nodes N, the parent function p, and the labeling function l, and write $T = (N, p, l)$. The children of a node $n \in N$ are denoted by $C(n)$.

We begin by defining the tree edit operations that we consider. Since there are many ways to transform one tree to another using these edit operations, we define a cost model for these edit operations, and then define the problem of finding a minimum-cost edit script that transforms one tree to another.

5.2.1 Edit Operations and Edit Scripts

In the following, we will assume that an edit operation e is applied to $T_1 = (N_1, p_1, l_1)$, and produces the tree $T_2 = (N_2, p_2, l_2)$. We write this as $T_1 \xrightarrow{e} T_2$. We consider the following six edit operations:

Insertion: Intuitively, an insertion operation creates a new tree node with a given label, and places it at a given position in the tree. The position of the new node n in the tree is specified by giving its parent node p and a subset C of the children of p. The result of this operation is that n is a child of p, and the nodes C, that were originally children of p, are now children of the newly inserted node n.

Formally, an insertion operation is denoted by $\text{INS}(n, v, p, C)$, where n is the (unique) identifier of the new node, v is the label of the new node, $p \in N_1$ is the node that is to be the parent of n, and $C \subseteq C(p)$ is the set of nodes that are to be the children of n. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup \{n\}$, $p_2(n) = p$, $p_2(c) = n, \forall c \in C$, $p_2(c) = p_1(c), \forall c \in N_1 - C$, $l_2(n) = v$, and $l_2(m) = l_1(m), \forall m \in N_1$.

Deletion: This operation is the inverse of the insertion operation. Intuitively, $del(n)$ causes n to disappear from the tree; the children of n are now the children of the (old) parent of n. The root of the tree cannot be deleted.

Formally, a deletion operation is denoted by $del(n)$, where $n \in N_1$ and n is not the root of T_1. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$ with $N_2 = N_1 - \{n\}$, $p_2(c) = p_1(n), \forall c \in C(n)$, $p_2(c) = p_1(c) \forall c \in N_2 - C(n)$, and $l_2(m) = l_1(m), \forall m \in N_2$.

Update: The operation $upd(n, v)$ changes the label of the node n to v.

Formally, an update operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $upd(n, v)$, where $n \in N_1$, and produces $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1$, $p_2 = p_1$, $l_2(n) = v$, and $l_2(m) = l_1(m), \forall m \in N_2 - \{n\}$.

Move: A move operation $\textsc{mov}(n, p)$ moves the subtree rooted at n to another position in the tree. The new position is specified by giving the new parent of the node, p. The root cannot be moved.

Formally, a move operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\textsc{mov}(n, p)$, where $n, p \in N_1$, and p is not in the subtree rooted at n. (The last restriction is necessary to disallow moving a subtree to a node in the same subtree, since the resulting structure would not be a tree.) The resulting tree is $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1$, $l_2 = l_1$, $p_2(n) = p$, and $p_2(c) = p_1(c), \forall c \in N_2 - \{n\}$.

Copy: A copy operation $\textsc{cpy}(m, p)$ copies the subtree rooted at n to a another position. The new position is specified by giving the node p that is to be the parent of the new copy. The root cannot be copied.

Formally, a copy operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\textsc{cpy}(n, p)$, where $n, p \in N_1$, and n is not the root. Let $T_3 = (N_3, p_3, l_3)$ be a new tree that is isomorphic to the subtree of T_1 rooted at n, and let n' be the root of T_3. The result of the copy operation is the tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup N_3$, $l_2(m) = l_1(m), \forall m \in N_1$, $l_2(m) = l_3(m), \forall c \in N_3$, $p_2(n') = p$, $p_2(m) = p_1(m), \forall m \in N_1$, and $p_2(m) = p_3(m), \forall m \in N_3$.

Glue: This operation is the inverse of a copy operation. Given two nodes n_1 and n_2 such that the subtrees rooted at n_1 and n_2 are isomorphic, $\textsc{glu}(n_1, n_2)$ causes the subtree rooted at n_1 to disappear. (It is conceptually "united" with the subtree rooted at n_2.) The root cannot be glued. Although the GLU operation may seem unusual, note that it is a natural choice for an edit operation given the existence of the CPY operation. As we will see in Example 5.2.1, inverting an edit script containing a CPY operations results in an edit script with a GLU operation. This symmetry in the structure of edit operations is useful in the design of our algorithms.

Formally, a glue operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\text{GLU}(n_1, n_2)$. Let T_3 be the subtree rooted at n_1, and let $T_4 = (N_4, p_4, l_4)$ be the subtree rooted at n_2. The precondition of this GLU operation is that T_4 is isomorphic to $T_3 - T_4$. The result of the glue operation is the tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 - N_4$, $p_2(c) = p_1(c), \forall c \in N_2$, and $l_2(c) = l_1(c), \forall c \in N_2$.

In addition to the above tree edit operations, one may wish to consider operations such as a *subtree delete* operation that deletes all nodes in a given subtree. Similarly, one could define a *subtree merge* operation that merges two or more subtrees. We do not consider such more complex edit operations in this chapter, but note that some of these operations, (e.g., subtree deletes) may be detected by post-processing the output of our algorithm.

We define an *edit script* to be a sequence of zero or more edit operations that can be applied in the order in which they occur in the sequence. That is, given a tree T_0, a sequence of edit operations $\mathcal{E} = e_1, e_2, \ldots, e_k$ is an edit script if there exist trees T_i, $1 \leq i \leq k$ such that $T_{i-1} \xrightarrow{e_i} T_i$, $1 \leq i \leq k$. We say that the edit script $\mathcal{E}$ transforms T_0 to T_k, and write $T_0 \xrightarrow{\mathcal{E}} T_k$.

Example 5.2.1 Consider the tree T_1 depicted in Figure 5.1. We represent the identifier of each node by the number inside the circle representing the node. The label of each node is depicted to the right of the node. Thus, the root of the tree T_1 has an identifier 1, and a label a. Figure 5.1 shows how T_1 is transformed by applying the edit script to $\mathcal{E}_1 = (\text{INS}(11, g, 1, \{9\}), \text{MOV}(2, 6), \text{CPY}(7, 1))$ T_1. Similarly, if we start with the tree T_2 in the figure, the edit script $\mathcal{E}_2 = (\text{GLU}(12, 7), \text{MOV}(2, 1), del(11))$ transforms it back to T_1. We write $T_1 \xrightarrow{\mathcal{E}_1} T_2$, and $T_2 \xrightarrow{\mathcal{E}_2} T_1$. □

When an edit script is applied to tree, as in Example 5.2.1, the node identifiers in the initial and final state of the tree determine a mapping between the nodes in the two states. Note however, that in an instance of a change detection problem, we are given two trees, without any correspondence between their node identifiers. That is, in a change detection problem involving the trees T_1 and T_2 of Figure 5.1, the node identifiers of T_2 would be unrelated to those of T_1. We will discuss this issue further in Section 5.3.

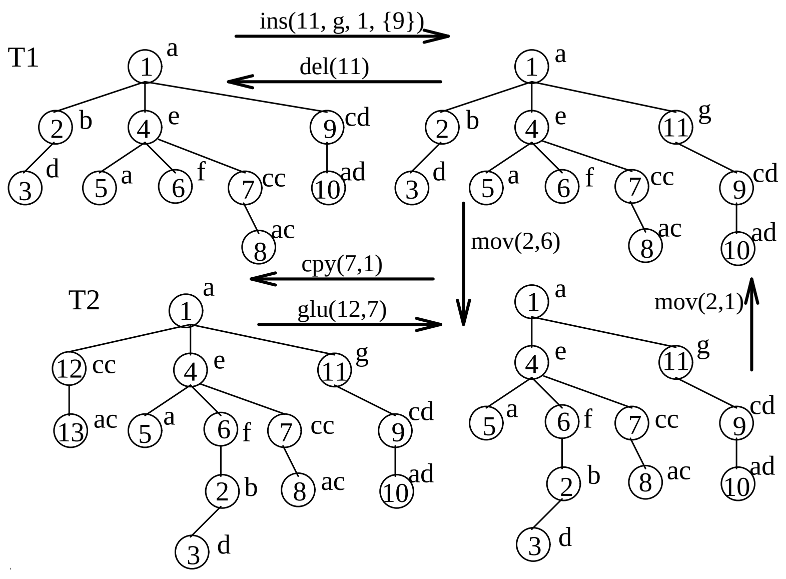


Figure 5.1: Edit operations on labeled trees

5.2.2 Cost Model

Given a pair of trees, there are, in general, several edit scripts that transform one tree to the other. For example, there is the trivial edit script that deletes all the nodes of one tree and then inserts all the nodes of the second tree. There are many other edit scripts that, informally, do more work than seems necessary. Formally, we would like to find an edit script that is "minimal" in the sense that it does no more work that what is absolutely required. To this end, we define a cost model for edit operations and edit scripts.

There are two major criteria for choosing a cost model. Firstly, the cost model should accurately capture the domain characteristics of the data being considered. For example, if we are comparing the schematics for two printed-circuit boards, we may prefer an edit script that has as few inserts as possible, and instead describes changes with moves and copies of the old components. However, if we are comparing text documents, we may prefer to see a paragraph as a new insertion, rather than a description of how it was assembled from bits and pieces of sentences from the old document. Secondly, the cost model should be simple to specify, and should require little effort from the user. For example, a cost model that requires the user to specify dozens of parameters is not desirable by this criterion, even though it may accurately model the domain.

Another issue is the trade-off between generality of the cost model and difficulty in computing a minimum-cost edit script. For example, a very general cost model would have a user-specified function to determine the cost of each edit operation, based on the type of the edit operation, as well as the particular nodes on which it operates. However, such a model is not amenable to the design of efficient algorithms for computing the minimum-cost edit script, since it does not permit us to reason about the relative costs of the possible edit operations.

With the above criteria in mind, we propose a simple cost model in which the costs of insertion, deletion, move, copy, and glue operations are given by constants, c_i, c_d, c_m, c_c, and c_g, respectively. Furthermore, given the symmetry between INS and DEL, and CPY and GLU, it is reasonable to use $c_i = c_d$, and $c_c = c_g$. Since, intuitively, a MOV operation causes a smaller change than either CPY or GLU, it is also reasonable to use $c_m < c_c$. Note, however, that our algorithms do not depend on these relationships between the cost parameters. The cost of an update operation depends on the old and new values of the label being updated; that is, $c(upd(n, v)) = c_u(v_0, v)$, where v_0 is the old label of n, and c_u is a domain-dependent function that returns a non-negative real number.

Finally, the *cost of an edit script* $\mathcal{E}$, denoted by $c(\mathcal{E})$, is defined as the sum of the costs of the edit operations in $\mathcal{E}$. That is, $c(\mathcal{E}) = \sum_{d \in \mathcal{E}} c(d)$.

Problem Statement: Given two rooted, labeled trees T_1 and T_2, find an edit script $\mathcal{E}$ such that $\mathcal{E}$ transforms T_1 to a tree that is isomorphic to T_2, and such that for every edit script $\mathcal{E}'$ with this property, $C(\mathcal{E}') \geq C(\mathcal{E})$.

5.3 Method Overview

In this section, we present an overview of algorithm MH-DIFF for computing a minimum-cost edit script between two trees. We present our algorithm informally using a

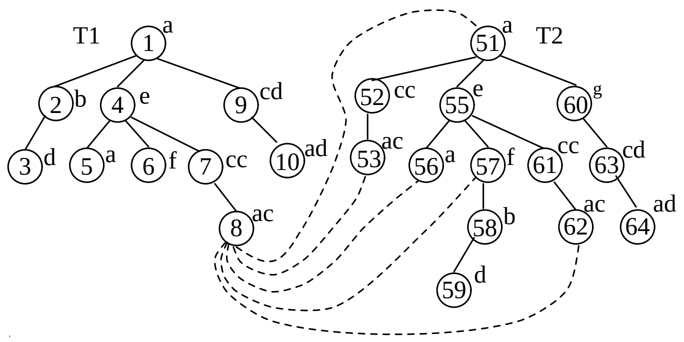


Figure 5.2: The trees for the running example in Section 5.3.

running example; the details are deferred to later sections.

Consider the two trees depicted in Figure 5.2. We would like to find a minimum-cost edit script that transforms tree T_1 into tree T_2. The reader may observe that these trees are isomorphic to the initial and final trees from Example 5.2.1 in Section 5.2. Note, however, that there is no correspondence between the node identifiers of T_1 and T_2 in Figure 5.2. This is because in Example 5.2.1 we applied a known edit script to a tree, transforming it to another tree in the process, whereas in this section, we are trying to find an edit script, given two trees with no information on the relationship between their nodes. Therefore, our first step consists of finding a correspondence between the nodes of the two given trees.

For example, consider the node 8 in Figure 5.2. We want to find the node in T_2 that corresponds to this node in T_1. The dashed lines in Figure 5.2 represent some of the possibilities. Intuitively, we can see that matching the node 8 to the node 51 does not seem like a good idea, since not only do the labels of the two nodes differ, but the two nodes also have very different locations in their respective trees; node 8 is a leaf node, while node 51 is the root node. Similarly, we may intuitively argue that matching node 8 to node 62 seems promising, since they are both leaf nodes and their labels match. However, note that matching a nodes based simply on their labels ignores the structure of the trees, and thus is not, in general, the best choice. We make this intuitive notion of a correspondence between nodes more precise below.

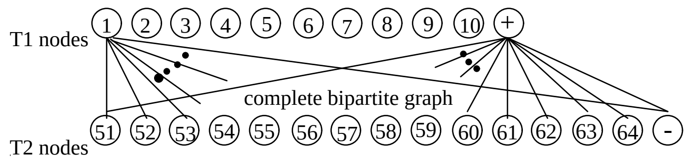


Figure 5.3: The Induced Graph for the trees in Figure 5.2

5.3.1 The Induced Graph

Consider the complete bipartite graph B depicted in Figure 5.3, consisting of the nodes of T_1 at the top, and the nodes of T_2 at the bottom, plus the special nodes $\oplus$ and $\ominus$. (For clarity, not all edges of the graph are shown in Figure 5.3.) We call B the *induced graph* of T_1 and T_2. The dashed lines in Figure 5.2 correspond to the edges of the induced graph. Intuitively, we would like to find a subset K of the edges of B that tells us the correspondence between the nodes of T_1 and T_2. If an edge connects a node $m \in T_1$ to a node $n \in T_2$, it means that n was "derived" from m. (For example, n may be a copy of m.) We say m is *matched* to n. A node matched to the special node $\oplus$ indicates that it was inserted, and a node matched to $\ominus$ indicates that it was deleted. Note that this matching between nodes need not be one-to-one; a node may be matched to more than one other nodes. (For example, referring to Figures 5.2 and 5.3, node 6 may be matched to both node 54 and node 59.) The only restriction is that a node be matched to at least one other node. Thus, finding the correspondence between the nodes of two trees consists essentially of finding an edge cover of their induced graph. (An edge cover of a graph is a subset K of the edges of the graph such that any node in the graph is incident on at least one edge in S.)

The induced graph has a large number of edge covers (this number being exponential in the number of nodes). However, we may intuitively observe that most of these possible edge covers of B are undesirable. For example, and edge cover that maps all nodes in T_1 to $\ominus$, and all nodes in T_2 to $\oplus$ seems like a bad choice, since it corresponds to deleting all the nodes of T_1 and then inserting all the nodes of T_2. We will define the correspondence between an edge cover of an induced graph and an edit script for the underlying trees formally in Section 5.4.2, where we also describe how to compute an edit script corresponding to an edge cover. For now, we simply note that, given an edge cover of the induced graph, we can compute a corresponding edit script for the underlying trees. Hence, we would like to select an edge cover of the induced graph that corresponds to a minimum-cost edit script.

5.3.2 Pruning the Induced Graph

We noted earlier that many of the potential edge covers of the induced graph are undesirable because they correspond to expensive and undesirable edit scripts. Intuitively, we may therefore expect a substantial number of the edges of the induced graph to be extraneous. Our next step, therefore, consists of removing (pruning) as many of these extraneous edges as possible from the induced graph, by using some *pruning rules*. The pruning rules that we use are *conservative*, meaning that they remove only those edges that we can be sure are not needed by a minimum-cost edit script. We discuss pruning rules in detail in Section 5.5.3, presenting only a simple example here.

As an example of the action of a simple pruning rule, consider the edge $e_1 = [5, 53]$, representing the correspondence between nodes 5 and 53 in Figure 5.2. Suppose that the cost $c_U(a, ac)$ of updating the label a of node 5 to the label ac of node 53 is 3 units. Furthermore, let the cost of inserting a node and deleting a node be 1 unit each. Then we can safely prune the edge $[5, 53]$ because, intuitively, given any edge cover K_1 that includes the edge e_1, we can generate another edge cover that excludes e_1, and that corresponds to an edit script that is at least as good as the one corresponding to K_1. As an illustration of such pruning, consider the edge cover $K_2 = K_1 - \{e\} \cup \{[5, \ominus], [\oplus, 53]\}$. This edge cover corresponds to an edit script that deletes the node 5, and inserts the node 53. These two operations cost a total of 2 units, which is less than the cost of the update operation suggested by the edge e in edge cover K_1. We therefore conclude that the edge $[5, 53]$ in our running example may safely be pruned. In Section 5.5.3 we present Pruning Rule 2, which is a generalization of this example.

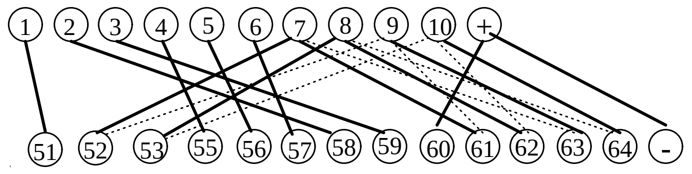


Figure 5.4: The induced graph of Figure 5.3 after pruning

5.3.3 Finding an Edge Cover

By applying the pruning rules to the induced graph of our running example (Section 5.5.3), say we obtain the *pruned induced graph* depicted in Figure 5.4. Although the pruned induced graph typically has far fewer edges than the original induced graph does, it typically still contains more edges than needed to form an edge cover. In Section 5.4.2 we will see that we need only consider edge covers that are *minimal*; that is, edge covers that are not proper supersets of another edge cover. In other words, we would like to remove from the pruned induced graph those edges that are not needed to cover nodes. For example, in the pruned induced graph shown in Figure 5.4, having all four of the edges $[7, 61]$, $[7, 63]$, $[9, 61]$, and $[9, 63]$ is unnecessary; we may remove either $[7, 63]$ and $[9, 61]$; or $[7, 61]$ and $[9, 63]$. However, it is not possible to decide a priori which of these options is the better one; that is, it is not obvious which choice would lead to an edit script of lower cost. With pruning, on the other hand, there was no doubt that certain edges could be removed.

One way to decide among these options is to enumerate all possible minimal edge covers of the pruned induced graph, find the edit script corresponding to each one (using the method described later in Section 5.4.2), and to pick the one with the least cost. However, given the exponentially large number of edge covers, this is obviously not an efficient algorithm. To compute an optimal edge cover efficiently, we need to be able to determine how much each edge in the edge cover contributes to the total cost of an edit script corresponding to an edge cover containing it. That is, we need to distribute the cost of the edit script corresponding to an edge cover over the individual edges of the edge cover. Once we have a cost defined for each edge

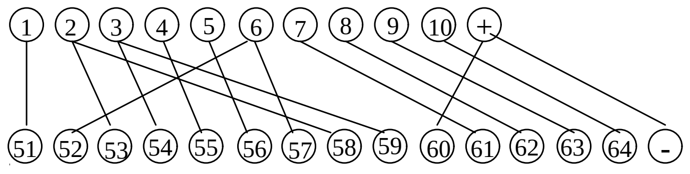


Figure 5.5: A minimum-cost edge cover of the induced graph in Figure 5.4

in the pruned induced graph, we can find a minimum-cost edge cover using standard techniques based on reducing the edge cover problem to a weighted matching problem [PS82, Law76]. For example, if the edges $[7, 61]$, $[7, 63]$, $[9, 61]$, and $[9, 63]$, have costs 0, 1.3, 0.2, and 2.4, respectively, then we generate an edge cover that includes $[7, 61]$ and $[9, 61]$, and excludes $[7, 63]$ and $[9, 61]$.

Note, however, that such a reduction of the edit script problem to an edge cover (and thus, weighted matching) problem cannot be exact, given the hardness of the edit script problem (unless $\mathcal{P} = \mathcal{NP}$, since we are considering a polynomial-time reduction). Indeed, our method of assigning costs to edges of the induced graph (Section 5.5.1) is only approximate, and thus the minimum-cost edge cover is not guaranteed to produce the best solution for the edit script problem.

5.3.4 Generating the Edit Script

Returning to the pruned induced graph of our running example, let us assume that we have gone through the process of determining the cost of each edge, and have computed a minimum-cost edge cover according to these costs, obtaining the edge cover depicted in Figure 5.5. Our next step consists of using this edge cover to compute an edit script that transforms the tree T_1 to the tree T_2. Our algorithm *CtoS* (Cover-to-Script) for this purpose is described in Section 5.5. Here, we briefly illustrate some of the ideas used by the algorithm by considering its action on an edge in the edge cover for our running example.

Consider the edge $e_1 = [7, 52]$ of the edge cover depicted in Figure 5.4. In Figure 5.6, we depict this edge in relation to the original trees. (We also depict two

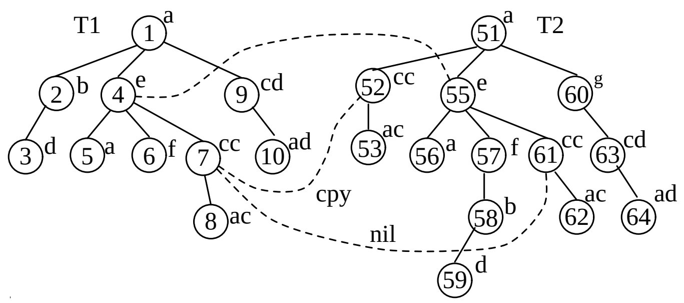


Figure 5.6: Annotating edges in the edge cover of Figure 5.5

other edges from the edge cover. The edge cover edges are shown as dashed lines in Figure 5.6. We observe that there is one other edge in the edge cover that is incident on node 7, viz. $[7, 61]$, suggesting that the node 7 was copied either directly, or indirectly (due to one of its ancestors being copied). Furthermore, we note that the parent (node 4) of node 7 is matched to the parent (node 55) of node 61 (i.e., the edge $[4, 55]$ exists in the edge cover), while the parent of node 52 is *not* matched to the parent of node 7. This matching of the parents suggests that node 61 is the original instance of node 7, while node 52 is the copy. We therefore generate a copy operation that copies the subtree rooted at node 7 to the location of node 52. A convenient way of depicting this copy operation is by *annotating* the corresponding edge ($[7, 52]$ in our example) with a CPY mark; this scheme allows us to talk about edit operations without having to refer to explicit node identifiers. Edges that do not correspond to any edit operation (e.g., $[6, 57]$ in our example) are annotated with a NIL mark. In the sequel, we will use such edge annotations interchangeably with the actual edit operations that they represent.

Consider next the edges $[8, 53]$ and $[8, 62]$. Although both these edge cover edges are incident on node 8, neither of them corresponds to a CPY operation, since the copy 52 of node 8 is generated "for free" when node 7 is copied. Therefore, both these edges are annotated NIL. Proceeding thusly, we annotate all the edges in the edge cover of our running example, to obtain the annotated edge cover depicted in Figure 5.7, which shows only the edges with non-nil annotations, for clarity. These

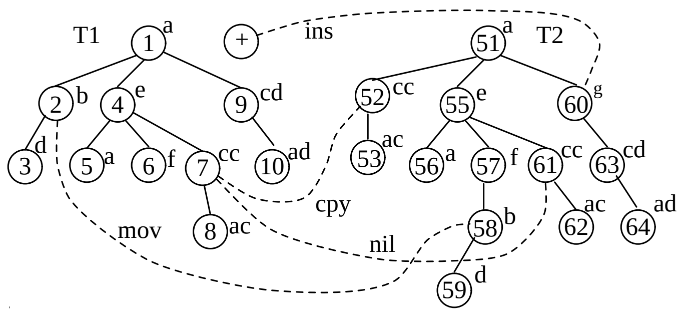


Figure 5.7: Annotated edges of the edge cover of Figure 5.5

annotations correspond to the following edit script:

$$(\text{INS}(g, 1, \{9\}), \text{MOV}(2, 6), \text{CPY}(7, 1))$$

We see that this edit script is identical to the one in Example 5.2.1, which happens to be a minimum cost edit script for our example. Of course, the above edit operations may also be listed in the following order

$$(\text{MOV}(2, 6), \text{CPY}(7, 1), \text{INS}(g, 1, \{9\}))$$

Both edit scripts have the same final effect, and have the same cost. In general, all edit scripts corresponding to a set of annotated edges have the same overall effect and the same cost.

For the above example MH-DIFF produces a minimum-cost edit script, but it may sometimes not find one with globally minimum cost. In Section 5.6 we evaluate how often this happens and we briefly discuss how one could perform additional searching in the neighborhood of the script found by MH-DIFF.

This concludes the overview of MH-DIFF. To summarize, the process consists of constructing an induced graph from the input trees, pruning the induced graph, finding a minimum-cost edge cover of the pruned induced graph, and finally, using this edge cover to obtain an edit script. In the following sections, we describe these phases in detail. For ease of presentation, we present these phases in a different order than the order in which they are performed. In particular, in Section 5.4, we begin by formally defining the correspondence between and edit script and an edge cover of the induced graph. In that section, we also describe the method for generating an edit script from an edge cover of the induced graph. In Section 5.5, we describe how the cost of an edit script is distributed over the edges of the corresponding edge cover of the induced graph. In that section, we also describe how this cost function is approximated by deriving upper and lower bounds on the cost of an edge of the induced graph, and how these bounds are used to prune the induced graph. Since finding a minimum-cost edge cover for a bipartite graph with fixed edge costs is a problem that has been previously studied in the literature [PS82, Law76], we do not present the details in this chapter.

5.4 Edge Covers and Edit Scripts

In this section, we describe algorithm *CtoS*, which generates an edit script between two trees, given an edge cover of their induced graph. Before we can describe this algorithm, we need to understand the relationship between an edit scripts between two trees and edge covers of their induced graph. Therefore, we first define the edge cover induced by an edit script. That is, we describe how, given an edit script between two trees, we generate an edge cover of the induced graph. (Note that this process is the reverse of the process the algorithm *CtoS* performs. However, a definition of this reverse process is needed for the description of the algorithm.)

5.4.1 Edge Cover Induced by an Edit Script

In Section 5.3, we introduced the graph induced by two trees T_1 and T_2 as the complete bipartite graph $B = (U, V, U \times V)$, with $U = N_1 \cup \{\oplus\}$ and $V = N_2 \cup \{\ominus\}$ (where N_1 and N_2 are the nodes of T_1 and T_2, respectively). Let $\mathcal{E}$ be an edit script that transforms T_1 to T_2; that is, $T_1 \xrightarrow{\mathcal{E}} T_2$. We now define the edge cover $K(\mathcal{E})$ induced by $\mathcal{E}$. Intuitively, we obtain $K(\mathcal{E})$ as follows. Create a copy T_3 of T_1, and introduce an edge between each node in T_1 and its copy in T_3. Apply the edit script to T_3, moving,

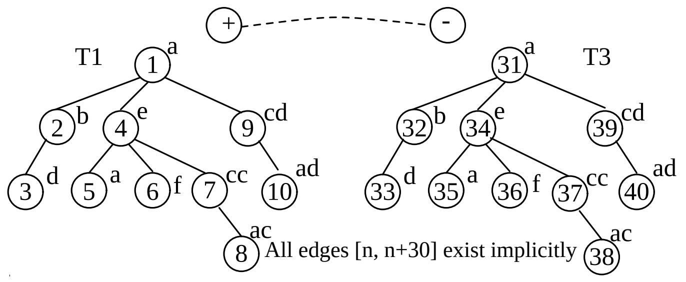


Figure 5.8: Example 5.4.1: the initial edge cover

copying, etc. the end-points of the edges with the nodes they are attached to as nodes are moved, copied, etc. Thus, when an a node $n \in T_3$ is copied, producing node n', any edge $[m, n]$ is split to produce an new edge $[m, n']$. The other edit operations are handled analogously. Furthermore, an edge between the special nodes $\oplus$ and $\ominus$ is added initially, and removed when it is no longer needed to cover either $\oplus$ or $\ominus$. The following example illustrates the above ideas.

Example 5.4.1 Consider the edit script from Example 5.2.1, and the initial tree T_1 from Figure 5.1. As described above, our first step consists of creating a copy T_3 of T_1, and adding an edge between each node of T_1 and its counterpart in T_3. We also add the special nodes $\oplus$ and $\ominus$, along with an edge connecting them. The result of this step is depicted in Figure 5.8. For clarity in presentation, the edges between the nodes of T_1 and their counterparts in T_3 are not shown in Figure 5.8; instead, we encode these edges using the node identifiers of T_1 and T_2. That is, as indicated in the figure, imagine an edge $[n, n + 30], \forall n = 1 \ldots 10$.

Our next step consists of applying the edit script from Example 5.2.1 to the tree T_3. To enable this application of the edit script for T_1 to T_3, we change the node identifiers in the edit script from the identifiers of the nodes of T_1 to those of T_3, obtaining $\mathcal{E}_1 = (\text{INS}(41, g, 31, \{39\}),\ \text{MOV}(32, 36),\ \text{CPY}(37, 31))$. As a result of the INS operation, a node with identifier 41 and label g is inserted as a child of node 31, and node 37 is made its child. In addition, we add an edge $[\oplus, 41]$ to the induced edge cover. Next, consider the action of the MOV operation, which moves node 32

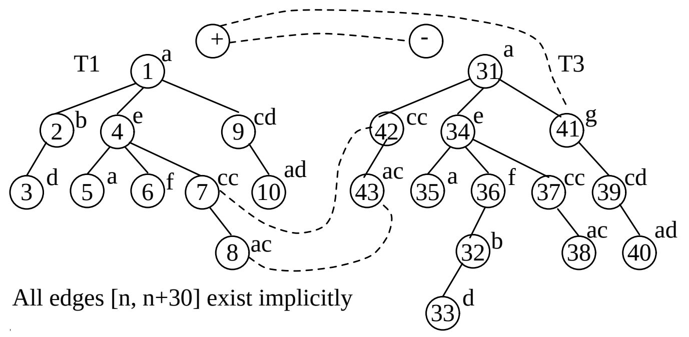


Figure 5.9: Example 5.4.1: the final edge cover

to become a child of node 37. This operation does not add any new edges to the edge cover. (The existing edges $[2, 32]$ and $[3, 33]$ continue to exist.) Finally, the CPY operation creates a copy of the subtree rooted at node 36, and inserts this copy as a child of node 31. In addition, the edges $[7, 42]$ and $[8, 43]$ are added to the edge cover. The result is depicted in Figure 5.9, (which also omits edges $[n, n + 30], \forall n = 1 \ldots 10$ for clarity). Note that the transformed tree T_3 is now isomorphic to the tree T_2 in Example 5.2.1, so that essentially, we now have an edge cover of the induced graph of T_1 and T_2.

$\square$

Let us now formalize the intuitive definition of the edge cover induced by an edit script presented in the above example. Let $\mathcal{E}$ be an edit script that transforms T_1 to T_2; that is, $T_1 \xrightarrow{\mathcal{E}} T_2$. We now define $K(\mathcal{E})$, the edge cover (of the induced graph of T_1 and T_2) induced by $\mathcal{E}$. Let T_3 be a tree that is isomorphic to T_1, with f being the isomorphism. Thus, $f : T_1 \rightarrow T_2$ is a one-to-one, onto function that preserves the parent-child and label relationships defining labeled trees. More precisely, $\mathit{label}(f(m)) = \mathit{label}(m)$, and $\mathit{parent}(f(m)) = f(\mathit{parent}(m))$ for all nodes $m \in T_1$. Let us extend f to $T_1 \cup \{\oplus\}$ and $T_2 \cup \{\ominus\}$ by defining $f(\oplus) = \ominus$). We will now define how, given the edit script $\mathcal{E}$, we derive a mapping $g(\mathcal{E})$, called the mapping induced by $\mathcal{E}$, from the isomorphism f. We will see that the mapping g is an onto mapping from T_1 to T_2, and is thus isomorphic to an edge cover of the induced graph B.

Base case: If the edit script is empty, that is if $\mathcal{E} = ()$, then $g = f$.
Inductive case: The edit script is non-empty. Let d be the last edit operation in the edit script $\mathcal{E}$; that is, $\mathcal{E} = \mathcal{E}'.d$ for some edit script $\mathcal{E}'$. Let T_2' be the tree script obtained by applying $\mathcal{E}$ to T_1; that is, $T_1 \xrightarrow{\mathcal{E}} T_2$. Let g' be (inductively) the mapping induced by $\mathcal{E}'$; that is $g' = g(\mathcal{E}')$. We have the following cases, based on the last edit operation d. (Recall the formal definitions of the edit operations from Section 5.2.)

Case 1: d is an update operation. Then $g(\mathcal{E}) = g(\mathcal{E}')$.

Case 2: d is an insert operation $\textsc{ins}(n, l, p, C)$. Then $g(\mathcal{E}) = g(\mathcal{E}') \cup \{(\oplus, n)\}$.

Case 3: d is a delete operation $del(n)$. If $n \in T_1$, then $g(\mathcal{E}) = g(\mathcal{E}') \cup \{(n, \ominus)\}$, else $g(\mathcal{E}) = g(\mathcal{E}')$.

Case 4: d is a move operation $\textsc{mov}(n_1, n_2)$. Then $g(\mathcal{E}) = g(\mathcal{E}')$.

Case 5: d is a copy operation $\textsc{cpy}(n_1, n_2)$. Let t_1 be the subtree rooted at n_1, and let t_1' be the subtree isomorphic to t_1 that is created as a result of this copy operation. Let h be the isomorphism between t_1 and t_1'. Then $g(\mathcal{E}) = g(\mathcal{E}') \cup h$.

Case 6: d is a glue operation $\textsc{glu}(n_1, n_2)$. Let t_1 be the subtree rooted at n_1, and let t_2 be the subtree (isomorphic to t_1) rooted at n_2. (Recall that the subtree t_1 disappears as a result of this glue operation, being "united" with the subtree t_2.) Let h be the isomorphism between t_1 and t_2. Let $h' = (n, g(\mathcal{E}')(n)) \forall n \in t_1$. Then $g(\mathcal{E}) = g(\mathcal{E}') \cup h - h'$.

Finally, if the $\oplus$ node and the $\ominus$ node are both mapped to more than one node, we remove $[\oplus, \ominus]$ from the mapping. Now observe that after performing the operations indicated above for all the edit operations in $\mathcal{E}$, T_3 is transformed to a tree that is isomorphic to T_2 (by the definition of $\mathcal{E}$), so that the mapping $g(\mathcal{E})$ may be viewed as an onto mapping between T_1 and T_2. An onto mapping between the nodes of T_1 and T_2 is isomorphic to an edge cover of the bipartite graph induced by T_1 and T_2; thus $g(\mathcal{E})$ defines the edge cover induced by an edit script.

5.4.2 Using Edge Covers

The goal of using an edge cover is that it should capture the essential aspects of an edit script; that is, no important information should be lost in going from an edit script to the edge cover induced by it. However, there are certain edit scripts for which this property does not hold. For example, consider an edit script $\mathcal{E}_2$ that inserts a node p as the parent of ten siblings (children of the same parent) $n_1, \ldots, n_{10}$, then moves p to another location in the tree, and finally deletes p. The node p is absent from both the initial tree and the final tree. Therefore, an edge cover of the initial and final trees contains no record of the temporary insertion of node p. Thus, we have lost some information in going from $\mathcal{E}_2$ to the edge cover.

Is the fact that our edge covers cannot capture edit scripts like $\mathcal{E}_2$ a problem? On the one hand, $\mathcal{E}_2$ could be the minimum cost edit script MH-DIFF is trying to find. For example, say that insert, delete, and move operations all cost one unit. The cost of $\mathcal{E}_2$ would then be the cost of one insert, plus the cost of one move, plus the cost of one delete, for a total cost of 3. If we do not use the "bulk move trick" that $\mathcal{E}_2$ uses, we need to move each of $n_1, \ldots, n_{10}$ individually, for a cost of 10. Thus, $\mathcal{E}_2$ could be the minimum cost edit script, and if we rule it out, then MH-DIFF would miss it.

On the other hand, scripts like $\mathcal{E}_2$ do not represent transformations that are meaningful or intuitive to an end user. In other words, if a user saw $\mathcal{E}_2$, he would not understand why node p was inserted, since it really has no function in his application. True, the costs provided by the user are intended to describe the desirability of edit operations, but if we abuse these numbers we can end up with "tricky" scripts like $\mathcal{E}_2$ that are more confusing than helpful.

Another example of a potentially unintuitive edit script is the following: Consider an edit script $\mathcal{E}_3$ that moves a node n_1 to become a child of another node n_2, then makes several copies of the subtree rooted at n_2 (thus making copies of n_1 as well), and finally deletes the original copy of n_1. This edit script moves n_1 to a place where it does not need to be (under n_2) only to generate free copies of n_1.

The cause of the unintuitive nature of the edit scripts described above is an interaction between different edit operations, which gives rise to a "compound" effect.

For example, in the edit script $\mathcal{E}_2$ above, the effect of the move operation is compounded because it acts on a node that was previously inserted. Similarly, in edit script $\mathcal{E}_3$ above, the effects of the copy operations are compounded because they act on a subtree into which a node was previously moved. Our approach is to disallow such unintuitive compound effects by restricting the interleaving of edit operations in an edit script. In particular, we require that edit operations be performed in a fixed order: deletes, copies, moves, updates, glues, inserts. This structuring requirement disallows tricks such as temporary insertion of a node, and complicated interleaving of move, copy, and glue operations.

Unfortunately, this structuring requirement also disallows some intuitive sequences of operations. For example, it does not allow an edit script that deletes a node produced as a result of a CPY operation. Thus, an edit script cannot copy a subtree containing 100 nodes if 99 of them are needed, because it would be unable to delete the unwanted copy of the 100th node. An analogous situation exists for INS and GLU operations. To mitigate this problem, we add a phase of deletions after copies, called *ghost deletions*. During this phase, only nodes produced by copy operations are permitted to be deleted. Furthermore, these deletions can act only on leaf nodes. (An interior nodes may be deleted only if all all its descendants are also deleted.) Analogously, we permit a phase of insertions, called *ghost insertions*, before the glue phase. Any nodes inserted in this phase must be removed by glue operations in the subsequent glue phase. Furthermore, nodes may be only inserted as leaves. (That is, interior nodes must be inserted before the insertion of their descendants.)

A reasonable restriction to impose on edit scripts is the following: Edit scripts may only copy subtrees from the original source. That is, instead of copying a subtree t' that was produced by copying some other subtree t, we copy t itself. We could express this restriction by disallowing copies of copied subtrees. However, such a restriction would have the following undesirable side-effect: If a subtree is copied to some node n, then from that point on none of the ancestors of n can be copied. We therefore rephrase our restriction as follows: When a subtree t is copied, any subtree t' it contains that was produced by copies are ignored. That is, the copy operation acts only on $t - t'$. We impose a symmetrical restriction on glue operations. Finally, just as we do not permit inserted nodes to be deleted, we disallow the gluing of nodes produced by copies.

More precisely, we define *structured edit scripts* to be edit scripts with the following properties: (1) No node is operated on by more than one structure-changing edit operation. (All edit operations except update are structure-changing.) (2) Edit operations are performed in phases, and the phases are ordered as follows: deletes, copies, moves, ghost deletes, updates, ghost inserts, glues, inserts. (3) In the ghost deletes phase, only nodes produced by copy operations are deleted. In the ghost inserts phase, only nodes that are later removed using glue operations are inserted. Ghost insertions and deletions operate only on leaf nodes. (4) A node produced by one copy operation is not copied by another. Similarly, a node that is the target of a glue operation is not the source of another glue operation. (5) A node produced by a copy operation is not glued.

We now describe how the above restrictions on structured edit scripts yield simplifications in the mapping between edge covers and scripts. A *minimal edge cover* is an edge cover that is not a proper superset of an edge cover. Minimal edge covers have the following useful characterization:

Lemma 4 *An edge cover is minimal if and only if it does not contain any path of length three.* □

Proof To see that a minimal edge cover K cannot contain a path of length three, suppose n_1, n_2, n_3, n_4 is a path in K. That is, $[n_i, n_{i+1}] \in K, i = 1 \ldots 3$. Then $K - \{[n_2, n_3]\}$ is an edge cover contradicting the minimality of K. Conversely, if K is non-minimal, there is some edge $[n_2, n_3] \in K$ such that $K' = K - \{[n_2, n_3]\}$ is an edge cover. Then K' contains at edges $[n_1, n_2]$ and $[n_3, n_4]$ for some n_1 and n_4, implying a three-path n_1, n_2, n_3, n_4 in K. □

Structured edit scripts have the following important property that allows us to consider only minimal edge covers in the rest of the chapter.

Lemma 5 *The edge cover induced by a structured edit script is a minimal edge cover (and thus does not contain a path of length three).* □

Proof Due to Lemma 4, it suffices to show that the edge cover induced by a structured edit script does not contain a three-path. Since the induced graph is bipartite, any three-path that is not incident on the special nodes $\oplus$ and $\ominus$. is of the form n', m, n, m' such that $m, m' \in T_1$ and $n, n' \in T_2$. Now the only edit operation that causes the induced cover to include more than one edge incident on a node $m \in T_1$ is copy. Therefore, m, n, and n' are acted on by some copy operation. We can similarly argue that m, n, and m' are acted on by some glue operation. However structured edit scripts cannot glue a node that has been acted on by a copy operation, thus implying no such three-path can exist in their induced edge covers. For the case of edges incident on the $\oplus$ and $\ominus$ nodes, we note that possibility of a three-path is avoided by the definition of the induced edge cover because it removes the edge $[\oplus, \ominus]$ whenever there are multiple edges incident on both $\oplus$ and $\ominus$. Thus there can be no three-path in the edge cover induced by a structured edit script. □

5.4.3 Generating an Edit Script from an Edge Cover

We now describe how, given a minimal edge cover K of the graph induced by trees T_1 and T_2, we compute a minimum-cost edit script corresponding to this edge cover. In particular, we present algorithm *CtoS*, for cover-to-script. The **input** to the algorithm consists of two rooted labeled trees T_1 and T_2 and and a minimal edge cover K of their induced graph $IG(T_1, T_2)$. As **output**, the algorithm CtoS produces an edit script $\mathcal{E}$ with the following properties:

1. $\mathcal{E}$ is a valid edit script with respect to T_1. That is, the operations in $\mathcal{E}$ can be applied to T_1 in order, and $\mathcal{E}$ is a *structured* edit script (see Section 5.4.2).

2. $\mathcal{E}$ transforms T_1 to a tree isomorphic to T_2. That is, $T_1 \xrightarrow{\mathcal{E}} T_2$.

3. The edge cover induced by $\mathcal{E}$ isomorphic to K.

4. There is no edit script with the above properties that has a lower cost than that of $\mathcal{E}$. That is, if $\mathcal{E}'$ is an edit script satisfying properties 1–3 above, then $c(\mathcal{E}') \geq c(\mathcal{E})$.

We now explain the **method** used by CtoS, summarized by the pseudo-code in Figures 5.10–5.20. The algorithm proceeds in *phases* that roughly reflect the phases of a structured edit script. The phases are named after the kinds of edit operations they produce. For example, the procedure phaseDel in Figure 5.10 describes the delete phase of the algorithm, which produces delete operations. A slight amendment to this rule is that the copy and glue phases, outlined in Figures 5.11 and 5.17 respectively, also produce some move operations. Intuitively, these are move operations used to take advantage of the free copies and glues described earlier.

We now describe some notation that is used by the pseudocode in Figures 5.10–5.20. As explained in Section 5.3, it is convenient to represent the edit operations in an edit script using an annotation on the corresponding edge of the edge cover. In the pseudo-code, these annotations are stored in sets P_n, where n is the name of the phase. For example, Pdel is the set of annotations representing delete operations. These sets are initialized to the empty set. Annotations are generated using a function *Annot*, which takes as argument details of the corresponding edit operation; each annotation thus generated has a unique identifier The variable `annNil` is initialized to a special annotation representing the null edit operation.

We refer to edges belonging to the given edge cover K as *K-edges*. We say two nodes are *matched* to each other if there is a K-edge connecting them. In order to simplify the pseudo-code, if K contains the special edge $[\oplus, \ominus]$, this edge is removed from K during initialization. For each tree node m, we keep track of the number of "free images of m" (i.e., copies of m obtained as a result of one of its ancestors being copied) in the set $m.F$. For each such free image, $m.F$ contains the annotation representing the edit operation that is responsible for the free image. Initially, $m.F$ is the singleton set {`annNil`} representing the original image of the node m in tree T_1.)

We use $E(x)$ do denote the set of K-edges that are incident on the node x. We use the function $nnda(m)$ to denote the nearest proper ancestor m' of m such that $[m', \ominus] \notin K$. Intuitively, this function returns the nearest non-deleted ancestor of a node in T_1. Analogously, we use the function $nnia(n)$ to denote the nearest proper ancestor n' of n such that $[\oplus, n'] \notin K$. Intuitively, this function returns the nearest

```
procedure phaseDel(tree T1, tree T2, cover K) {
    for each edge $e = [m, \ominus] \in K$ do {
        ann1 $\leftarrow$ Annot(del(m));
        e.respAnn $\leftarrow$ annNil;
        Pdel $\leftarrow$ Pdel $\cup$ { ann1 };
        partner $\leftarrow$ partner $-$ { (m, $\ominus$) };
        m.F := $\emptyset$;
    }
}
```

Figure 5.10: CtoS: generating delete operations

non-inserted ancestor of a node in T_2. We also use the function $nnid(n)$ to denote the set of nearest non-inserted descendants of a node n in T_2. More precisely, we have the following, where p is the parent function for tree T_2:

$$nnid(n) = \{n_1 | \exists j \in Z^+ : p^j(n_1) = n \wedge \forall i = 1, \ldots, j-1 : [\oplus, p^i(n_1)] \in K\}$$

We define a relation *partner* between the nodes of $T_1^+ = T_1 \cup \{\oplus\}$ and $T_2^+ = T_2 \cup \{\ominus\}$. Initially, *partner* is the relation defined by the given edge cover K, less the special edge $[\oplus, \ominus]$. As the algorithm proceeds, the *partner* relation is modified in such a way that when the algorithm terminates, *partner* is an isomorphism between the tree T_f obtained by applying the generated edit script to a working tree T_3 that is isomorphic to T_1, and T_2. We use *partners(m)* to denote the set of nodes n such that $(m, n) \in partner$. The pseudo-code for CtoS uses a function $pickPartner(m)$ to intuitively denote the unique, final partner of a node m. More precisely, if $partners(m)$ is a singleton set $\{n\}$, *pickPartner* returns n; otherwise *pickPartner* returns a placeholder string "pp(m)" that will be replaced by the partner of m (guaranteed to be unique) when the algorithm terminates. We generalize *pickPartner* to a set M of nodes in T_1 as follows: $pickPartner(M) = \{pickPartner(m) \mid m \in M\}$.

The first phase of the algorithm is the delete phase (Figure 5.10), in which we generate an edit operation DEL(m) for each node m that is matched to the special node $\ominus$. We claim that any edit script that matches m to $\ominus$ must contain this *del*

```
procedure phaseCpy(tree T1, tree T2, cover K) {
    for each node $m \in T1$, in pre-order, do {
        if($|E(m)| > 1$) {
            for each edge e1 = [m, n1] $\in$ E(m) such that
                    [nnda(m), nnia(n1)] $\in$ K, do {
                e2 $\leftarrow$ [nnda(m), nnia(n1)];
                ann2 $\leftarrow$ e2.respAnn;
                if(ann2 $\in$ m.F) doCNil(e1, e2);
                else doMovOrCpy(e1);
            }
            for each edge e1 = [m, n1] $\in$ E(m) such that
                    [nnda(m), nnia(n1)] $\notin$ K, do {
                doMovOrCpy(e1);
            }
        }
        else if($[m, \ominus] \notin K$) {
            let $E(m)$ = { e1 };
            e1.respAnn $\leftarrow$ annNil;
            m.F $\leftarrow$ m.F $-$ { annNil };
        }
        if($[m, \ominus] \notin K$) {
            for each annotation ann2 $\in$ m.F do {
                ann1 $\leftarrow$ Annot(del(m.ann2));
                Pgdel $\leftarrow$ Pgdel $\cup$ { ann1 } $-$ { ann2 };
            }
        }
    }
}
```

Figure 5.11: CtoS: generating copy-related operations

operation, due to the following observations: Firstly, any node matched to $\ominus$ is absent from the final tree. Furthermore, there are only two ways in which a node can be made to disappear: either it is deleted explicitly, or it is glued to some other node. (We use here the fact that structured edit scripts cannot first glue a node to another and then delete the second node.) However, the second method will not result in m matching $\ominus$ in the edge cover induced by the script; instead, m will match the node to which it was glued. Therefore we can safely produce a DEL(m) operation for all such nodes m.

```
procedure doCNil(edge e1, edge e2) {
    let e1 = [m, n1];
    ann2 ← e2.respAnn;
    m.F ← −= { ann2 };
    e1.respAnn ← ann2;
    if(ann2 ≠ annNil) {
        partner ← partner −= { (m, n1) } + { (m.ann2, n1) };
    }
}
```

Figure 5.12: CtoS: bookkeeping for free copies

The next phase of the algorithm, summarized in Figure 5.11, generates copy operations and move operations used to correctly position copies. In particular, it looks for sets two or more of K-edges incident on a common node $m \in T_1$. Note that from Lemma 5, and the observation that minimal edge covers cannot contain any path of length three, it follows that if $e = [m, n]$ is such an edge, there can be no other K-edge incident on n. We call such a set of edges a flower with base m. This set of edges represents copies of the node m. However, as we have seen in Section 7.1.2, some of the copies of m could be produced as a result of some ancestor of m being copied. We call such copies *free copies* of m. Our algorithm considers flowers in preorder of the base nodes. As copy operations are generated for some node m, we also keep track of the number of free copies of nodes in the copied subtree. Knowing the number of available free copies allows us to determine exactly which flowers correspond to explicit copy operations and which correspond to implicit (free) copies. Furthermore, any unused free copies are nodes that need to be deleted after the copy operation is performed. These are the ghost deletions we introduced above. Finally, note that a free copy may need to be moved to its final location; this situation is easily detected by checking whether the parents of the affected nodes match.

The update phase of the algorithm is straightforward, and produces an update operation for each edge $[m, n]$ such that the labels of m and n differ. Since we are considering only structured edit scripts, there is no way to avoid such an update; in

```
procedure doMovOrCpy(edge e1) {
    let e1 = [m, n1];
    if($|m.F| > 1$) {
        ann2 ← pick(m.F);
        m.F ← m.F − { ann2 };
        ann1 ← Annot(mov(m.ann2, pickPartner(nnia(n1))));
        e1.respAnn ← ann2;
        Pmov ← + { ann1 };
        if(ann2 ≠ annNil) {
            partner ← partner −= { (m, n1) } ∪ { (m.ann2, n1) };
        }
    }
    else {
        ann1 ← Annot(cpy(m, pickPartner(nnia(n1))));
        e1.respAnn ← ann1;
        Pcpy ← Pcpy ∪ { ann1 };
        partner ← partner − { (m, n1) } ∪ { (m.ann2, n1) };
        updFSets(ann1, m);
    }
}
```

Figure 5.13: CtoS: finding spare images for copy

```
procedure updFSets(Annot ann1, node m) {
    for each child c of m do updFSetsAux(ann1, c);
}
procedure updFSetsAux(Annot ann1, node m) {
    if($[m, \ominus] \notin K$ and $[\oplus, n] \notin K$) m.F ← m.F ∪ { ann1 };
    for each child c of m do updFSetsAux(ann1, c);
}
```

Figure 5.14: CtoS: bookkeeping for free images

```
procedure phaseMov(tree T1, tree T2, cover K) {
    for each edge e = [m, n] ∈ K such that
            $|E(m)| = |E(n)| = 1$ do {
        if([nnda(m), nnia(n)] ∈ K) ; else {
            ann1 ← Annot(mov(m, pickPartner(nnia(n)))));
            Pmov ← ∪ { ann1 };
        }
        e.respAnn ← annNil;
        m.F ← m.F − { annNil };
        n.F ← m.F − { annNil };
    }
}
```

Figure 5.15: CtoS: generating move operations

```
procedure phaseUpd(tree T1, tree T2, cover K) {
    for each pair (m, n) ∈ partners such that
            $m \neq \oplus$, $n \neq \ominus$, and $l(m) \neq l(n)$ do {
        ann1 ← Annot(upd(m, l(n)));
        Pupd ← ∪ { ann1 };
    }
}
```

Figure 5.16: CtoS: generating update operations

particular, tricks like updating a node and then copying it are disallowed.

The glue phase of the algorithm, summarized by the pseudo-code in Figures 5.17, 5.18, and 5.19, is analogous to the copy phase. A notable difference is the use of a function *correspPartner(n,a)* to determine the node m in T_1 that is matched by a K-edge to n, and that is glued by the annotation a. We can compute this node m using the bookkeeping relation gr maintained by the glue phase. More precisely, as we argue below, the set $P = \{x \mid gr(x, n, a)$ is guaranteed to be the singleton $\{m, n, a\}$. Similarly, the insert phase, summarized in Figure 5.20, is analogous to the delete phase. Intuitively, the only major difference between these phases and the earlier copy and delete phases is that these phases perform actions from T_2's point of view instead of T_1's; it is thus useful to consider them "mirror images" of the earlier phases.

Given the sets of annotations produced by algorithm CtoS, the final edit script is produced by generating the edit operation suggested by each annotation, and then ordering these operations in phases as required by our definition of structured edit scripts earlier. Within each phase, edit operations are ordered as follows: Delete, ghost delete, move, and glue operations are ordered using any bottom-up order for T_1. Insert, ghost insert, and copy operations are ordered using any top-down order for T_2. Update operations are ordered arbitrarily.

Discussion: In the rest of this section, we argue that the edge cover produced by algorithm CtoS satisfies the four properties listed earlier.

The **edge cover induced by an edit script $\mathcal{E}$** is defined operationally as follows: We start with T_1 and a working tree T_3 isomorphic to T_1. The initial working set of edges K_w contains edges corresponding to the isomorphism I_1 between T_1 and T_3. We apply the edit script $\mathcal{E}$ to T_3, resulting in a sequence of trees $T_4, T_5, \ldots, T_f$. When a node is inserted in the (regular) inserts phase, we match it to $\oplus$; when a node is inserted in the ghost inserts phase, we do not add any edges to K_w. When a node is deleted in the (regular) deletes phase, we redirect any K_w-edges incident on it to the special node $\ominus$; when a node is deleted in the ghost deletes phase, we simply remove any K_w-edges incident on it. When a node is updated or a subtree is moved, there is no change in in K_w. When a subtree is copied, we add to K_w an edge from each

```
procedure phaseGlu(tree T1, tree T2, cover K) {
    for each node $n \in T2$, in pre-order, do {
        if($|E(n)| > 1$) {
            for each edge e1 = [m1, n] ∈ E(n) such that
                    [nnda(m1), nnia(n)] ∈ K, do {
                e2 ← [nnda(m1), nnia(n)];
                ann2 ← e2.respAnn;
                if(ann2 ∈ n.F) doGNil(e1, e2);
                else doMovOrGlu(e1);
            }
            for each edge e1 = [m1, n] ∈ E(n) such that
                    [nnda(m1), nnia(n)] ∉ K, do {
                doMovOrGlu(e1);
            }
        }
        else if($[\oplus, n] \notin K$) {
            let $E(n)$ = e1;
            e1.respAnn ← annNil;
            n.F ← n.F − { annNil };
        }
        if($[\oplus, n] \notin K$) {
            for each annotation ann2 ∈ n.F do {
                ann1 ← Annot(ins(n*,l(n), correspPartner(p(n),ann2)));
                Pgins ← Pgins ∪ { ann1 };
                gr ← gr ∪ { (n*, n, ann2) };
            }
        }
    }
}
```

Figure 5.17: CtoS: generating glue-related operations

```
procedure doGNil(edge e1, edge e2) {
    let e1 = [m1, n];
    ann2 $\leftarrow$ e2.respAnn;
    n.F $\leftarrow$ n.F $-$ { ann2 };
    e1.respAnn $\leftarrow$ ann2;
    gr $\leftarrow$ gr $\cup$ { (m1, n, e1.respAnn) };
    if(ann2 $\neq$ annNil) {
        partner $\leftarrow$ partner $-$ { (m1, n) };
    }
}
```

Figure 5.18: CtoS: bookkeeping for free glues

```
procedure doMovOrGlu(edge e1) {
    let e1 = [m1, n];
    if($|n.F| > 1$) {
        ann2 $\leftarrow$ pick(n.F);
        n.F $\leftarrow$ n.F $-$ { ann2 };
        ann1 $\leftarrow$ Annot(mov(m1, correspPartner(nnia(n), ann2)));
        Pmov $\leftarrow$ Pmov $\cup$ { ann1 };
        e1.respAnn $\leftarrow$ ann2;
        gr $\leftarrow$ $\cup$ { (m1, n, e1.respAnn) };
        if(ann2 $\neq$ annNil) {
            partner $\leftarrow$ partner $-$ { (m1, n) };
        }
    }
    else {
        ann1 $\leftarrow$ Annot(glu(m1, pickPartner(n)));
        Pglu $\leftarrow$ Pglu $\cup$ { ann1 };
        e1.respAnn $\leftarrow$ ann1;
        partner $\leftarrow$ partner $-$ { (m1, n) };
        updFSets(ann1, n);
    }
}
```

Figure 5.19: CtoS: finding spare images for glue

```
procedure phaseIns(tree T1, tree T2, cover K) {
    for each edge $e = [\oplus, n] \in K$ do {
        ann1 ← Annot(ins(n*, l(n), pickPartner(p(n)), pickPartner(nnid(n))));
        e.respAnn ← annNil;
        Pins ← Pins ∪ { ann1 };
        partner ← partner − { (⊕, n) } ∪ { (n*, n) };
        n.F ← ∅;
    }
}
```

Figure 5.20: CtoS: generating insert operations

newly created node to the partner of its original copy. When a subtree is glued, we redirect edges incident on each node in the subtree that disappears (the "source" of the glue operation) to that node's counterpart in the subtree that remains (the target of the glue operation). Finally, if either of $\oplus$ and $\ominus$ is exposed (not covered by the edges in K_w), we add the edge $[\oplus, \ominus]$ to K_w. The final set of edges thus obtained is an edge cover of the induced graph of T_1 and T_f, $IG(T_1, T_f)$. We call this set of edges the edge cover induced by $\mathcal{E}$, and denote it by $K_f = K(\mathcal{E})$.

An edge cover K is **minimal** if no proper subset of K is an edge cover. It is easy to verify that a minimal edge cover does not contain any path of length three. We further require that a minimal edge cover of $IG(T_1, T_2)$ not contain any paths of length two incident on either of the special nodes $\ominus$ and $\oplus$.

In the following discussion, we use the following notational convention: A primed node m_i' represents a node in some working tree $T_j, j \in [3, f]$ such that if $m_i' \in T_3$ then there is a node $m_i \in T_1$ such that $I_1(m_i, m_i')$. (Recall that I_1 is the isomorphism between T_1 and T_3. Note that if $m_i' \notin T_3$, there is no requirement regarding the existence of a node $m_i \in T_1$.)

Property 1: Valid Structured Edit Script: We now argue that the edit script $\mathcal{E}$ produced by the algorithm is a valid structured edit script for the tree T_1. In particular, we show that each edit operation is valid, and that the edit script is structured. In the following discussion of edit operations, we use T to refer to the working tree at the time an edit operation is applied. That is, for an edit operation $d \in \mathcal{E}$, T is the tree obtained by applying $\mathcal{E}'$ to T_3, where T_3 is a tree isomorphic to T_1, and $\mathcal{E}'$ is the prefix of $\mathcal{E}$ up to, and not including, d.

Consider first any **delete** operation produced by the algorithm in procedure `phaseDel`. This operation is of the form $del(m')$. (The pseudocode in Figure 5.10 assumes that the edit script operates on the tree T_1, whereas we argue based on its operation on the isomorphic tree T_3; we thus replace each reference to $m \in T_1$ by $m' \in T_3$.) In order to be valid, the identifier m' must refer to a node that exists in T. Clearly, since m exists in T_1, and since T_3 is isomorphic to T_1, m' exists in T_3. Only delete and glue operations make nodes disappear. Glue operations are performed after deletes; therefore m' cannot disappear due to a glue operation. Furthermore, m' cannot disappear due to some other delete operation because `phaseDel` considers each node in T_1 at most once. Therefore, the delete operation is valid.

Next, consider any **copy** operation produced by the algorithm in procedure `phaseCpy`. This operation is of the form $cpy(m', pickPartner(nnia(n_1)))$. In order to be valid, m' and $pickPartner(nnia(n_1))$ must refer to nodes that exist in T. By the reasoning used for delete operations above, we can argue that m' exists in T. We assume, without loss of generality, that K matches the root of T_1 uniquely to the root of T_2, and vice versa. Therefore, $nnia(n_1)$ exists in T_2. Let $pickPartner(nnia(n_1)) = m_2'$. By the definition of $nnia(n_1)$, m_2' cannot be an inserted node. If $m_2 \in T_1$, we know $m_2' \in T$ because no node that has a partner in T_2 is ever deleted by `phaseDel`, and because there are no glue operations before copy operations. Suppose m_2' is produced as a result of a copy operation. By the top-down ordering of the copy phase (procedure `orderOps`), m_2' is produced before this copy operation is applied. Furthermore, m_2' cannot subsequently be deleted (since there are no deletes between copy operations) or be glued (since glue operations are performed after copy operations) before this copy operation. Therefore m_2' exists in T and this copy operation is valid.

Now consider any **move** operation produced by the algorithm in procedure `doMovOrCpy`. This operation is of the form $mov(m'.ann2, pickPartner(nnia(n1)))$. As above, we can argue that $pickPartner(nnia(n1)))$ exists in T. Note that $ann2$

is obtained from $m.F$. If $ann2$ is the null annotation `annNil`, $m'.ann2$ refers to the original node m'; we can argue, as above, that m' is not removed by any previous edit operation. If $ann2$ is not null, we note that the only time an annotation is added to $m.F$ is when one of the ancestors of m' is marked to be copied, thus implying an indirect copy of m'; $m'.ann2$ denotes this copy. Now, $m'.ann2$ cannot be glued before this move operation because all glue operations are performed after all move and copy operations. Furthermore, $m'.ann2$ cannot be deleted because ghost deletes are performed after moves. Thus, $m'.ann2$ exists in T, and this move operation is valid.

Using arguments similar to those above, it is easy to establish that the ghost delete operations produced by `phaseGlu`, the update operations produced by `phaseUpd`, and the move operations produced by `phaseMov` are valid.

Now consider any **glue** operation produced by the algorithm in procedure `doMovOrGlu`. This operation is of the form $glu(m'_1, pickPartner(nnia(n)))$. As above, we can argue that m'_1 and $m'_2 = pickPartner(nnia(n)))$ exist in T. In order for this glue operation to be valid, the subtree $st(m'_1)$ rooted at m'_1 must be isomorphic to the subtree $st(m'_2)$ rooted at m'_2; we now demonstrate this isomorphism.

In particular, we show that $st(m'_1)$ and $st(m'_2)$ are both isomorphic to a specially constructed tree, $nist(n)$. Intuitively, $nist(n)$ is obtained from $st(n)$ by "short circuiting" any nodes in $st(n)$ that are matched to $\oplus$. More formally we define the nodes, parent function, and label function of $nist(n)$ as follows: $nist(n) = (N', p', l')$ where $N' = \{n' \in st(n) | [\oplus, n'] \notin K\}$, and $\forall n' \in nist(n)$, $p'(n') = nnia(n')$ and $l'(n') = l(n')$. Let T_g be the working tree just before the glue phase, and let p_g be its parent function.

Consider the edge $e_1 = [m_1, n]$; let $e_1.respAnn = \alpha$. Consider the relation gr constructed in `phaseGlu`. It is easy to observe that for each node $y \in nist(n)$ there is exactly one node $x \in st(m'_1)$ such that $(x, n, \alpha) \in gr$. We therefore define a function $g : nist(n) \to st(m'_1)$ as follows: $g(y) = x$ where $(x, y, \alpha) \in gr$. We know that $p_g(m'_i) = nnda(m_i)'$ for each $m'_i \in T_g$, and $p'(y) = nnia(y)$ for all $y \in T_2$. Therefore, it is easy to see that g preserves the parent and label functions.

Let us now show that g is a one-to-one function. Consider a node $y \in nist(n)$. If $g(y)$ is an inserted node y^*, clearly y^* is not referenced by gr again, and thus $y^* \neq g(y')$ for $y' \neq y$. If $g(y)$ is a node m_i' such that $[m_i, n] \in K$, note that $|E(n)| > 1$ implies $|E(m_i)| = 1$ (due to minimality of K), so that there can be no other edge in K incident on m_i, and therefore m_i cannot be $g(y')$ for any $y' \neq y$. Thus g is a one-to-one function from $nist(n)$ to $st(m_1')$ that preserves the parent and label functions.

Let us now show that g maps $nist(n)$ onto $st(m_1')$; that is, we show that for each $x \in st(m_1')$, there is a $y \in nist(n)$ such that $g(y) = x$. If x is an inserted node, it must be ghost inserted, since regular insertions are performed after the glue phase. Thus (x, y, α) is is added to gr for some y and α when x is inserted, and we have $g(y) = x$. If x is produced by a copy, x will be skipped when this glue operation is performed due to `detectGluSkips`. If x is not produced by an insertion or a copy operation, then $x = m_3'$ such that $[m_3, n_3] \in K$ for some $n_3 \in T_2$. It is easy to see that $[p_g(m_3'), p'(n_3)] \in K$. Since we know $m_3' \in st(m_1')$ and $[m_1, n]$ is the only edge in K that is incident on m_1', it follows that $n_3 \in nist(n)$, implying that (x, n, α') is added to gr when node n_3 is handled by `phaseGlu`. We thus have $g(n_3) = x$, showing that g is onto. We have thus shown that g is a one-to-one, onto function from $nist(n)$ to $st(m_1')$ that preserves the parent and label functions; therefore $st(m_1')$ is isomorphic to $nist(n)$. We can argue analogously that $st(m_2')$ is also isomorphic to $nist(n)$, so that $st(m_1')$ and $st(m_2')$ are isomorphic, and the glue operation is valid.

Using an argument analogous to that used for `phaseMovOrCpy`, we can show that the move operations in `doMovorGlu` are valid. Now consider an insert operation produced by `phaseIns`. This operation is of the form $ins(n^*, l(n), pickPartner(p(n)),$ $pickPartner(nnid(n)))$. Trivially, we avoid using existing identifiers for the newly created nodes as n^*, and the label $l(n)$ is valid. Let $pickPartner(p(n)) = m_1'$. If m_1' is not an inserted node, we can argue, as we have done while discussing other edit operations, that $m_1' \in T$. If m_1' is an inserted node, we note that insert operations are performed in a top-down T_2 order, so that m_1' is inserted before this insert operation. Thus, $m_1' \in T$ in both cases. We can argue as before that each node $x \in pickPartner(nnid(n))$ exists in T. The only remaining condition for validity of this insert operation is that in T, $p(x) = m_1'$ for all $x \in pickPartner(nnid(n))$. In the discussion of Property 2, we will show that for $m \in T_1$, $n \in T_2$ such that $[m, n] \in K$, we have $partner(p_i(m), nnia(n))$, where p_i is the parent function of the tree T_i just before the insert phase. Therefore, in T_i we have $p_i(x) = pickPartner(nnia(n))$ for all $x \in pickPartner(nnid(n))$. Now if $nnia(n) = p(n)$, we have $m'_1 = pickPartner(nnia(n))$ giving the required result $p(x) = m'_1$. If $nnia(n) \neq p(n)$, the insertion of nodes corresponding to nodes between $nnia(n)$ an n proceeds in a top-down manner, and each inserted node has all nodes in $pickPartner(nnid(n))$ as its children. Thus, when n^* is inserted, $p(x) = m'_1$. Thus this insert operation is valid. The validity of insert operations produced in `phaseGIns` follows by a similar argument.

We have shown above that each edit operation produced by the algorithm is valid. We now show that the resulting edit script is a structured edit script. The procedure `orderOps` ensures that the edit operations are performed in the order required of structured scripts. All the insert operations in `Pgins` produced by `phaseGlu` insert leaf nodes since the last argument (set of children) is $\emptyset$. It is easy to verify that any node deleted in `Pgdel` is a leaf node because if any of its original children are not deleted, then they are moved to another location before the delete operation. The top-down by T_2 ordering of copy operations, and procedure `detectCpySkips` ensure that a copied subtree is not further copied; similarly, glued subtrees are not subjected to further glue operations. Furthermore, no node produced by by a copy operation is ever glued because of `detectCpySkips` and `detectGluSkips`. Therefore, the edit script generated by the algorithm is a valid structured edit script.

Property 2: Transformation $T_1 \xrightarrow{\mathcal{E}} T_2$**:** We now show that the edit script $\mathcal{E}$ produced by the algorithm transforms T_3 (a tree isomorphic to T_1) to a tree T_f that is isomorphic to T_2. In particular, we show that the *partner* relation is an isomorphism between T_f and T_2. We begin by showing that *partner* is a one-to-one relation between T_f and T_2. First, observe that the algorithm never adds a partner to a node that already has one or more partners; similarly, the algorithm never removes a partner for a node that has exactly one partner without also adding another partner for it. Thus, once a node has exactly one partner at some stage in the edit script, it will continue to have exactly one partner for the rest of the script. In particular, this fact implies that nodes in T_3 that have exactly one partner, also have exactly one partner in T_f. A similar argument holds for nodes in T_2. Consider a node $m' \in T_3$ such that $[m, \ominus] \in K$; m' is deleted and is thus absent from T_f. Now consider a node $n \in T_2$ such that $[\oplus, n] \in K$; for such a node n, the procedure `phaseIns` produces an insert operation to create a node n^* which then becomes the sole partner of n.

Now consider a node m' in T_3 that originally has multiple partners. In procedure `phaseCpy`, all of m''s partners except those connected to m via an edge with `respAnn = annNil` are removed. Observe further that there is always exactly one edge with this property incident on m (because an edge $[m, n1]$ is assigned a `respAnn` of `annNil` either because it is the only edge incident on m, or because the edge $[nnda(m), nnia(n1)]$ has `respAnn = annNil`). Therefore it follows that all but one partners of m' are removed by procedure `phaseCpy`. An analogous argument shows that all but one partners of a node n in T_2 that originally has multiple partners are removed by procedure `phaseGlu`. (In this case, we initially have $partner(m_i, n) \forall [m_i, n] \in E(n)$. In `phaseGlu`, (m_i, n) is removed from $partner$ for all $e_i = [m_i, n] \in E(n)$ except when `ei.respAnn = annNil`. As before, we argue that there is exactly one edge in $E(n)$ with this property; say that edge is $[m_2, n]$. Then after the glue phase, we have $partner(n) = m_2$.) Finally, note that the partner relation is initially complete (that is, every node in $T_1 \cup T_2$ has at least one partner), and further, whenever new nodes are added, they are assigned a partner. Therefore, the partner relation between the final tree T_f and the tree T_2 is also complete. Thus we have shown that $partner$ is a **one-to-one** and **complete** relation between T_f and T_2.

We now show that the partner relation **preserves the parent function**; that is, $p_f(partner(n)) = partner(p(n))$ for all nodes $n \in T_2$, where p_f is the parent function of T_f. First, note the following property, which is easily verified from the actions of the algorithm: If $[m, n] \in K$, $m \in T_1$, $n \in T_2$, and $m' \in T_f$, then $partner(m', n)$ when the algorithm terminates. (Recall that $m' \in T_3$ is a node such that $I_1(m, m')$ where I_1 is the isomorphism between T_1 and T_3.) Note also that the only three ways of changing the parent of an existing node are (1) moving the node, (2) deleting its parent, and (3) inserting a node as its parent.

Case 1: $[\oplus, n] \in K$: In this case, procedure `phaseIns` changes the partner of n from $\oplus$ to m', where m' is a newly created node. We see that the parent of this node is the (final) partner of the node $p(n)$; that is, $partner(n) = m'$ and $p_f(m') = partner(p(n))$ as required.

Case 2: $\exists m_1 : [m_1, n] \in K, m_1 \neq \oplus$: Let T_i denote the working tree immediately before the insert phase, and let p_i be the parent function for T_i. In each of the three sub-cases below, we show below that $\exists m'_2 : partner(m'_2, n) \wedge partner(p_i(m'_2), nnia(n))$. Then, if $p(n) = nnia(n)$, we have $p_i(m'_2) = p_f(m'_2)$, which gives us the required result. Otherwise, $[\oplus, p(n)] \in K$, so that `phaseIns` produces an insert operation creating a node m'_3 such that $partner(m'_3, p(n))$. Since $n \in nnid(y)$ for all y between $p(n)$ and $nnia(n)$ (inclusive), $partner(n)$ (i.e., m'_1 is made a child of m'_3, giving $p(m_1) = partner(p(n))$ as required. In the following, assume for now that $nnda(m_1)'$ is not glued (directly or indirectly) implying that if m'_1 is not moved, deleted, or glued, then $p_i(m'_1) = nnda(m_1)'$. (We will do away with this assumption later.)

Case 2.1: $|E(m_1)| = |E(n)| = 1$: Consider the two possibilities in procedure `phaseMov`. If m_1 is moved, its new parent is the final partner of $nnia(n)$; it is easy to verify that the parent just before the insert phase, $p_i(m_1)$, is also the final partner of $nnia(n)$, giving $partner(p_i(m_1), nnia(n))$ as required. If m_1 is not moved, we know $[nnda(m_1), nnia(n)] \in K$, implying $partner(nnda(m_1)', nnia(n))$ (due to R1). Now since all nodes between m'_1 and $nnda(m_1)'$ are deleted due to `phaseDel`, we have $p_i(m'_1) = nnda(m_1)'$. Thus we have the required $partner(p_i(m'_1), nnia(n))$.

Case 2.2: $|E(m_1)| > 1, |E(n)| = 1$: In this case, (m'_2, n) (for some node m'_2) is added to the partner relation in either (1) procedure `doCNil` or (2) procedure `doMovOrCpy`. In scenario (1), $[nnda(m), nnia(n)] \in K$ which, since $nnda(m_1)' = p_i(m'_1)$, gives $partner(p_i(m'_1), nnia(n))$ as needed. In scenario (2), m'_1 is either moved to, or created by a copy at, $pickPartner(nnia(n))$; thus $partner(p_i(m'_2), nnia(n))$ as needed.

Case 2.3: $|E(m_1)| = 1, |E(n)| > 1$: After the glue phase, $partner(n) = m_2$, where $[m_2, n]$ is the unique edge in $E(n)$ with `respAnn = annNil`. We note that if `phaseGlu` invokes `doGNil`, $[nnda(m_1), nnia(n)] \in K$, implying $partner(nnda(m_1)', nnia(n))$, which gives $partner(p_i(m'_2), nnia(n))$ (using $p_i(m'_1) = nnda(m_1)'$). If `phaseGlu` invokes `doMovOrGlu`, $p_i(m'_2) = correspPartner(nnia(n), \texttt{annNil}) = partner(nnia(n))$;

Thus $partner(p_i(m'_2), nnia(n))$ in this case too.

Now let us return to our assumption that $nnda(m_1)'$ is not glued and show that it is not necessary for our argument. If $nnda(m_1)'$ disappears due to a glue operation, we know that in the subtree that is the target of the glue operation, there is a node m'_3 such that $st(nnda(m_1)')$ is isomorphic to $st(m'_3)$. Therefore $st(m'_3)$ contains a node m'_4 such that $p(m'_4) = m'_3$. Furthermore, since $st(m'_3)$ is the target of the glue operation, $partner(m'_4, n)$, and $partner(m'_3, nnia(n))$; thus m'_3 and m'_4 can replace $nnda(m_1)'$ and m'_1 (respectively) in the above argument.

The $partner$ relation also **preserves the label function**; that is, $l(partner(m)) = l(m)$ for all nodes $m \in T_1 \cup T_2$. This fact follows easily from the action of procedure `phaseUpd`. We have thus demonstrated a relation $partner$ between T_f and T_2 that is one-to-one and complete, and that preserves the parent and label functions; that is, partner is the required isomorphism between T_f and T_2.

Property 3: Induced Cover: We now show that the edge cover $K_f = K(\mathcal{E})$ (of the Induced Graph of T_1 and T_f) induced by the edit script $\mathcal{E}$ produced by the algorithm is isomorphic to the given edge cover K (of the Induced Graph of T_1 and T_2). We first show that for each edge $e = [m_1, n_1] \in K$ ($m_1 \in T_1^+$, $n_1 \in T_2^+$), there exists a corresponding edge $e' = [m_1, partner(n_1)] \in K_f$ such that $partner(n_1) \in T_f$, where we define $partner(\oplus, \oplus)$ and $partner(\ominus, \ominus)$ for notational convenience.

The edge $e_s = [\oplus, \ominus]$ is a special case: It is easy to see that if K_f and K are isomorphic ignoring e_s, then they are also isomorphic considering e_s. If $n_1 = \ominus$ and $m_1 \neq \oplus$, the node m'_1 is deleted by $\mathcal{E}$, due to procedure `phaseDel`. Therefore, $[m_1, \ominus] \in K_f$, giving $[m_1, partner(n_1)] \in K_f$ (using $partner(\ominus) = \ominus$). If $m_1 = \oplus$ and $n_1 \neq \ominus$, a node m'_1 is inserted by $\mathcal{E}$ due to procedure `phaseIns`. Therefore, $[\oplus, m'_1] \in K_f$. Since `phaseIns` also ensures $partner(m'_1, n_1)$, we have $[m_1, partner(n_1)] \in K_f$ as needed. Given the above, in the rest of this discussion of Property 3 we assume that $m_1 \neq \oplus$ and $n_1 \neq \ominus$.

If $|E(m_1)| = |E(n_1)| = 1$, we have $partner(m'_1, n_1)$ due to procedure `phaseMov`. We know that $[m_1, m'_1] \in K_3$. If $[m_1, m'_1] \in K_f$, we have the required result. If not, it must be the case that m'_1 disappears at some stage in the edit script. A node can disappear in only two ways: Either it is deleted or it is glued. Since K is minimal, m_1

is not matched to $\ominus$; therefore `phaseDel` does not generate a delete operation for m'_1. Furthermore, `phaseCpy` deletes only nodes produced as a result of a copy operation, and therefore cannot delete m'_1. In `phaseGlu` glue operations are produced only for edges with $|E(n)| > 1$, which excludes m'_1 from being glued directly. Suppose m'_1 is glued indirectly due to a glue operation acting on one of its ancestors. If this case, the edge $[m_1, m'_1]$ gets transferred to $[m_1, m'_2]$, where m'_2 is the node corresponding to m'_1 in the subtree that is the target of the glue operation; furthermore, we have $m'_2 = \mathit{partner}(n_1)$, so that m'_2 effectively takes the place of m'_1 in our argument. Thus, $[m_1, \mathit{partner}(n_1)] \in K_f$ as needed.

If $|E(m_1)| > 1$ and $|E(n_1)| = 1$, $\mathit{partner}(n_1)$ is assigned some node m'_2 in `phaseCpy`. The node m'_2 is obtained from the pool of copies of m'_1, as accounted for in the set $m.F$. (The node m'_2 may be either the original m'_1, or a copy of m'_1 produced by a copy operation acting on m'_1 or one of its ancestors.) Since a copy of a node receives copies of all edges incident on the original, $[m_1, m'_1] \in K_3$, and no node that is removed from $m.F$ is ever deleted or glued, it follows that $[m_1, m'_2] \in K_f$. Since $m'_2 = \mathit{partner}(n_1)$, we have $[m_1, \mathit{partner}(n_1)] \in K_f$ as required.

If $|E(m_1)| = 1$ and $|E(n_1)| > 1$, `phaseGlu` sets $\mathit{partner}(n_1)$ to the node m'_3 where $[m_3, n_1]$ is the unique (see the discussion of Property 2) edge incident on n_1 with `respAnn = annNil`, which implies that m'_3 is never the source of a glue operation. Furthermore, since there are no deletes after glues, $m'_3 \in T_f$ giving $[m_1, m'_3] \in K'$ as needed.

Thus, we have shown that for every edge in K, there is a corresponding edge in K_f. We can argue in the reverse direction in an analogous manner; that is, we can show that for every edge $[m_1, m'_2] \in K_f$, there is an edge $[m_1, \mathit{partner}(m'_2)] \in K$. Thus we conclude that K and K_f are isomorphic.

Property 4: Minimum Cost: We now show that there can be no edit script with Properties 1–3 above that costs less than the edit script $\mathcal{E}$ produced by our algorithm. Let $\mathcal{E}'$ be any edit script that satisfies Properties 1–3. From the definition of an induced edge cover, it follows that $\mathcal{E}'$ must produce an insert operation for each edge $[\oplus, n_1] \in K$; similarly, it must produce a delete operation for each edge $[m_1, \ominus] \in K$. Therefore, $\mathcal{E}'$ cannot have any fewer **insert** or **delete** operations than $\mathcal{E}$. Furthermore since the deletion of a node m_2' present in the initial tree T_3 produces an edge $[m_2, \ominus]$ in the induced script, $\mathcal{E}'$ cannot delete any nodes that are present in the original tree other nodes m_i' such that $[m_i, \ominus] \in K$. Similarly, since the insertion of a node m_3' that remains in the final tree T_f results in an edge $[\oplus, m_3']$ in the induced cover, $\mathcal{E}'$ cannot insert any nodes that remain in the final tree other than those matched to some node $n_2 \in T_2$ such that $[\oplus, n_2] \in K$. (Thus, the only insert operations that $\mathcal{E}'$ may have that are not in $\mathcal{E}$ are ghost inserts; similarly the only delete operations that $\mathcal{E}'$ may have that are not in $\mathcal{E}$ are ghost deletes.) Let I_2' be the isomorphism between $T_f' = \mathcal{E}'(T_3)$ and T_2.

Let us now consider the **copy** operations produced by the algorithm. From the definition of the edge cover induced by an edit script, and from the fact that $K(\mathcal{E}')$ is isomorphic to K, we have the following: If $E(m_1) = \{e_i\}_{i=1}^{k}$ where $e_i = [m_1, n_i]$, then $\mathcal{E}'$ must copy m_1' (directly or indirectly) at least $k-1$ times. Since $\mathcal{E}'$ performs copy operations before any move operations, the only way an indirect copy of m_1' can be made is by copying a node m_2' such that m_2 is an ancestor of m_1 in T_1. Structured edit scripts cannot delete or glue a node that is either the source of, or is produced by, a direct copy operation. Therefore, an edit script cannot make unneeded copies of a node for the purpose of creating copies of nodes in its subtree since the surplus copies cannot be removed. Therefore, if m_1' is a node that is indirectly copied j times, then there is some $m_2 \in T_1$ such that m_2 is an ancestor of m_1 and $|E(m_2)| = j + 1$. Thus, the number of times a node m_1' is copied indirectly is $ic(m_1') = \max\{|E(x)| - 1 : anc(x, m_1)\}$. It is easy to see that the algorithm counts exactly $ic(m_1') + 1$ using the set $m_1.F$; that is, $ic(m_1') = |m_1.F| - 1$. Therefore, $\mathcal{E}'$ must produce all the copy operations produced by the algorithm in `phaseCpy`.

Let $\mathcal{E}$ be an edit script that transforms T_3 to T_f. Let I_1 be the isomorphism between T_1 and T_3, and let I_2 be the isomorphism between T_f and T_2. Let $copied(x, y)$ denote that the node y is produced by copying the node x (directly or indirectly). By considering the effect of each edit operation in an edit script on its induced edge cover, we can establish the following result:

R1: For all $m_1 \in T_1$ and $n_1 \in T_2$, $[m_1, n_1] \in K(\mathcal{E})$ if and only if one of the following holds true: (1) $\exists m_1' \in T_f : I_1(m_1, m_1') \wedge I_2(m_1', n_1)$; (2) $\exists m_1' \in T_3, m_2' \in T_f :$

$copied(m_1', m_2') \wedge I_1(m_1, m_1') \wedge I_2(m_2', n_1)$.

Let us now consider **move** operations. The only operations that change the parent of an existing node are move, and interior node insertion and deletion. However, the deletion of an interior node x changes the parent function in a restricted manner: The new parent of each of its children $C(x)$ is set to x's parent before the deletion. Similarly, the insertion of an interior node y changes the parent function in a restricted manner: Each of y's new children are required to be children of y's parent before the insertion.

Consider nodes $m_1' \in T_3$ and $n_1 \in T_2$ such that $I_2'(m_1', n_1)$. In the working tree T_d just after the deletion phase, $p_d(m_1') = nnda(m_1)'$. Suppose m_1' is not moved by $\mathcal{E}'$; then in the working tree T_i just before the insertion phase, we have $p_i(m_1') = nnda(m_1)'$ because there are no interior node inserts or deletes between T_d and T_i.

Case 1: The partner of $p(n_1)$ is not inserted; that is, $p(n_1) = nnia(n_1)$. Isomorphism requires that $I_2'(p(m_1'), p(n_1))$, that is, $I_2'(nnda(m_1)', nnia(n_1))$, which implies $[nnda(m_1), nnia(n_1)] \in K(\mathcal{E}')$, due to R1.

Case 2: The partner of $p(n_1)$ is inserted; that is $p(n_1)$ is a proper descendant of $nnia(n_1)$, and $I_2'(m_2', p(n_1))$ for some inserted node m_2'. Let $anc(x, y)$ denote that x is an ancestor of y. Let m_k' be the inserted node such that $I_2(m_k', n_k)$, where $anc(n_k, n_1)$ and $p(n_k) = nnia(n_1)$. Observe that the truth value of $(x, y) \in anc$ is not affected by the insertion of a node $z \neq x, y$. Therefore, our choice of n_k and the fact that I_2' preserves the anc relation implies $anc(m_k', m_1')$. Now m_k' cannot be an ancestor of $nnda(m_1)'$ because if it were, $nnda(m_1)'$ would have to match a node between n_1 and $nnia(n_1)$, which is impossible since only inserted nodes can match nodes between n_1 and $nnia(n_1)$. Furthermore, no node can exist (in the current or any later working tree) between m_k' and $nnda(m_1)'$ because such a node would have to match a node between n_k and $p(n_k)$, which is absurd. Therefore, $p(m_k') = nnda(m_1)'$, so that $I_2'(p(m_k'), p(n_k))$ gives $I_2(nnda(m_1)', nnia(n_1))$. Using R1, we thus have $[nnda(m_1), nnia(n_1)] \in K(\mathcal{E}')$.

We have shown that for each pair of nodes $m_1' \in T_3$, $n_1 \in T_2$ such that $I_2(m_1', n_1)$, if m_1' is not moved, $[nnda(m_1), nnia(n_1)] \in K(\mathcal{E}')$. It follows that a move operation is required for each edge $[m, n] \in K$ such that $[nnda(m), nnia(n)] \notin K$. Thus, for each move operation produced by `phaseMov`, there must be a corresponding move operation in $\mathcal{E}'$. Similarly, we can argue that the move operations produced in `phaseMovOrCpy` or `phaseMovOrGlu` must also be produced by $\mathcal{E}'$. (A subtlety is that the choice, if any, of which surplus copies of a node are to be deleted, and thus which copies are to retained, and moved as needed, is immaterial for the cost of an edit script since ghost deletions of a node require that all its children be deleted or moved earlier in the script; an analogous situation exists for ghost insertions.)

Let us now consider **update** operations. Consider an edge $[m, n] \in K$ such that $m \in T_1$, $n \in T_2$, and $l(m) \neq l(n)$. Using R1, we have one of the following three cases: (1) $|E(m)| = |E(n)| = 1$ and $I_2'(m', n)$. Clearly this requires an update operation to change the label of m'. (2) $|E(n)| = 1$, $E(m) = \{[m, n_i]\}_{i=1}^{k}$, and there are k nodes $x_1, \ldots, x_k$ such that $x_i = m' \vee \mathit{copied}(m', x_i)$ and $I_2'(x_i, n_i)$. In this case, since $l(x_i) = l(m) \neq l(n)$, we need k update operations to change the labels of x_i. (Note that since copies are done before updates, the label of each x_i must be updated separately, even if there exist $n_i, n_j, i \neq j$ such that $l(n_i) = l(n_j)$.) (3) $|E(m)| = 1$, $E(n) = \{[m_i, n]\}_{i=1}^{k}$, and all but one of the nodes m_i' are glued to some node m_p' such that $I_2'(m_p', n)$. If $m = m_p$, clearly m' must be updated. If $m \neq m_p$, m' is glued to m_p', so that m' must be identical to the label of m_p'. Now if $l(m_p') = l(n)$, clearly m' must be updated. If $l(m_p') = l(m') \neq l(n)$, we note that both m_p' and m' must be updated to $l(n)$ because there can be no updates after a glue operation. Thus, in all cases, we have shown that each edge $[m, n] \in K$ such that $m \in T_1$, $n \in T_2$, and $l(m) \neq l(n)$ necessitates an update operation. By considering the action of previous phases of the algorithm on the partner relation, it is easy to verify these are exactly the update operations produced by our algorithm in `phaseUpd`. Thus $\mathcal{E}'$ has no fewer update operations than $\mathcal{E}$.

Finally, we can argue using the ideas described above that $\mathcal{E}'$ must include all the glue and move operations produced by the algorithm in `phaseGlu`. Thus we conclude that $\mathcal{E}'$ contains all the edit operations in $\mathcal{E}$, and therefore costs no less than $\mathcal{E}$.

5.5 Finding the Edge Cover

In this section we describe how MH-DIFF finds a minimal edge cover of the induced graph. The resulting cover will serve as input to algorithm *CtoS* (Section 5.4). Our goal is to find not just any minimal edge cover, but one that corresponds to a minimum-cost edit script. Let us call such an minimal edge cover the *target cover*.

Consider an edge e in our pruned induced graph. To get to the target cover, MH-DIFF must decide whether e should be included in the cover. To reach this decision, it would be nice if MH-DIFF knew the "cost" of e. That is, if e remains in the target cover, then it would be annotated (by algorithm *CtoS*) with some operation, and we could say that the cost of this operation is the cost of e. Unfortunately, we have a "chicken and the egg problem" here: *CtoS* cannot run until we have the target cover, and we cannot get the target cover until we know the costs it will imply. To break the impasse, our approach uses the following idea:

Instead of trying to compute the actual cost of e, we compute an upper and lower bound to this cost. These bounds can be computed without the knowledge of which other edges are included in the target cover, and serve two purposes: Firstly, they allow us to design pruning rules that are used to conservatively eliminate unnecessary edges from the induced graph. Secondly, after pruning, the bounds can guide our search for the target cover.

As an enhancement, we actually use a variation on the edge cost suggested above. The following example shows that simply "charging" each annotation to the edge it is on is not entirely "fair." We are given a tree T_1 containing two nodes, n_1 and n_2 with the same label l. Furthermore n_1 has children n_{11} and n_{12} with labels a and b, respectively, and n_2 has children n_{21} and n_{22} with labels c and d, respectively. Suppose T_2 is a logical copy of T_1. (That is, T_1 and T_2 are isomorphic.) Consider an edge cover that matches each node in T_1 to its copy in T_2 except that it "cross matches" n_1 and n_2 across the trees, as shown in Figure 5.21. Given this edge cover, algorithm *CtoS* will produce a move operation for each of the nodes n_{11}, n_{12}, n_{21}, and n_{22}. However, these move operations were caused not by any mismatching of the nodes n_{11}, n_{12}, n_{21}, or n_{22}, but instead, by the mismatching of n_1 and n_2. Therefore it would be

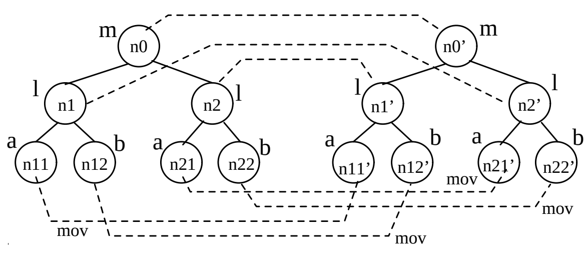


Figure 5.21: Distributing edge costs fairly

intuitively more fair to charge these move operations to the edges responsible for the mismatch, viz. $[n_1, n_2']$ and $[n_2, n_1']$. To achieve this, we use the following scheme: If e is annotated with INS, DEL, or *upd* in the target cover, we do charge e for this operation. However, if e is annotated by MOV, CPY, or GLU, then the *parent* of e, and not e is charged. We call the edge costs computed in such a fashion *fair costs.*

In summary, MH-DIFF first computes upper and lower bounds for the fair cost of each edge in the pruned induced graph. These bounds are then used to prune edges in the induced graph, and finally to search for the target cover. We begin by defining the fair cost of an edge below.

5.5.1 An Edge-wise Cost Function

Let K be an *annotated* minimal edge cover. For an edge $e \in K$, if the annotation on e is MOV, CPY, or GLU, let $c_x(e)$ denote the cost of that operation. If e is annotated with INS, DEL, or UPD, then let $c_s(e)$ denote the cost of the operation. Furthermore, let $E(m)$ be the set of edges in K that are incident on m, that is, $E(m) = \{[m, n] \in K\}$. Let $C(m)$ be the set of the children of m. We then define the *fair cost* of each edge $[m, n] \in K$ as follows:

$$\begin{aligned} c_K([m,n]) &= c_s(m,n) \\ &+ \frac{1}{2|E(m)|} \sum_{m' \in C(m)} \sum_{[m',n'] \in K} c_x([m',n']) \end{aligned}$$

$$+ \quad \frac{1}{2|E(n)|} \sum_{n' \in C(n)} \sum_{[m',n'] \in K} c_x([m',n']) \tag{5.1}$$

Note that this cost depends on K, and thus is not a function of e alone. The following lemma states that the above scheme of distributing the cost of an edge cover over its component edges is a sound one; that is, adding up the cost edge-wise yields the overall cost of the edge cover.

Lemma 6 *If K is an annotated, minimal edge cover of the graph induced by two trees, then $c(K) = \sum_{e \in K} c_K(e)$.* □

Proof By accounting. Recall that the cost $c(K)$ of an annotated edit script is the sum of the costs of the annotations in K (where the cost of each annotation is equal to the cost of the edit operation it represents). Each annotation in K is on some edge $e \in K$. If the annotation is an *upd*, it is charged (by $c_K(e)$) to the edge e itself. For other annotations, each node of e is charged for half the cost of the annotation. Furthermore, the cost of each node is distributed evenly over all edges $e' \in K$ incident on its parent. Since the special edge between the (dummy) roots of the two trees being considered is never annotated (without loss of generality), it follows that the two methods of accounting for the cost of an annotated edge cover are equivalent. □

5.5.2 Bounds on Edge Costs

Although Lemma 6 suggests a method of distributing the cost of an annotated edge cover (and thus an edit script) over the component edges, the cost of each edge depends on the other edges present in the edge cover, and is thus not directly useful for computing a minimum-cost edge cover. However, we use that distribution scheme to derive upper and lower bounds on the fair cost $c_K(e)$ of an edge e over all minimal edge covers K.

Intuitively, given that the cost of any *upd* annotation on an edge is charged to that edge (by Equation 5.1), a simple choice for the lower bound on the cost of an edge $[m,n]$ is simply the cost $c_u(m,n)$ of updating the label m to that of n. However, we can do a little better. In some cases, selecting an edge $[m,n]$ (as part of the edge cover being constructed) may *force* some of the children m' of m to be moved to n. In particular, this happens for those children of m' for which there is no edge that could possibly match m' to a child of n. We call such moves *forced moves.* In cases where we can determine a forced move exists, the cost of a MOV is added to the lower bound cost. However, according to Equation 5.1 not all the cost of a forced move goes to edge $[m, n]$. In the worst case, the number of edges incident on m, $|E(m)|$, is large, leaving $[m, n]$ with an insignificant contribution. However, if $|E(m)|$ is greater than 1, we know by Lemma 5 that $|E(n)| = 1$, so forced moves on the n side would contribute to $[m, n]$. Thus, we may add the minimum of the second and the third terms in Equation 5.1 to the lower bound function.

Formally, let E be the set of edges in the induced graph of T_1 and T_2. (As we will see later, although E initially includes all edges in the complete bipartite graph, the pruning of edges results in successive reduction of the size of E.) For notational convenience, let us also define $c_w(m, n)$ to be $c_u(m, n)$ if m and n are regular nodes, 0 if $(m = \oplus) \wedge (n = \ominus)$, c_i if$(m = \oplus) \wedge (n \neq \ominus)$, and c_d if $(m \neq \oplus) \wedge (n = \ominus)$. Further, we define the *forced move cost*, $c_{mf}(m', n)$ of a node $m' \in T_1$ with respect to another node $n \in T_2$ as follows: $c_{mf}(m', n) = c_m$, if $\not\exists n' \in C(n)$ such that $[m', n'] \in E$, and 0 otherwise. The cost $c_{mf}(m, n')$ is defined analogously. The *lower bound fair cost*, c_{lb}, of an edge can then be expressed as follows:

$$
\begin{aligned}
c_{lb}([m,n]) = {} & c_w(m,n) + \\
& \frac{1}{2}\min\left\{\sum_{m'\in C(m)} c_{mf}(m',n),\ \sum_{n'\in C(n)} c_{mf}(m,n')\right\}
\end{aligned}
$$

For notational convenience in defining the upper bound, let us now define a *conditional move cost*, c_{mc}. Intuitively, $c_{mc}(m', n)$ costs one MOV cost unless there is a partner of m' that is a child of n. Formally, $c_{mc}(m', n) = 0$, if $\exists n' \in C(n)$ such that $[m', n'] \in E$, and c_m otherwise. The cost $c_{mc}(n', m)$ is defined analogously. Using reasoning similar to that used for deriving the lower bound cost above, we arrive at the following definition for the *upper bound fair cost*, c_{ub}, of an edge:

$$c_{ub}([m,n]) \;=\; c_w(m,n)$$

$$+ \frac{1}{2} \sum_{m' \in C(m)} (c_c(|E(m')| - 1) + c_{mc}(m', n))$$
$$+ \frac{1}{2} \sum_{n' \in C(n)} (c_g(|E(n')| - 1) + c_{mc}(n', m))$$

Note that both $c_{ub}(e)$ and $c_{lb}(e)$ can be computed without knowledge of the target cover. Furthermore, the following lemma states that the above definitions of $c_{ub}(e)$ and $c_{lb}(e)$, are upper and lower bounds, respectively, on the fair cost contribution $c_K(e)$ of edge e to *any* minimal edge cover K that contains e.

Lemma 7 *Let $B = (U, V, E)$ be the bipartite graph induced by trees T_1 and T_2. Let $B' = (U, V, E')$, where $E' \subseteq E$. Let $\mathcal{K}$ denote the collection of all minimal edge covers of B'. We then have the following inequalities:*

$$c_{lb}(e) \leq \min_{K \in \mathcal{K}} c_K(e) \quad \text{and} \quad c_{ub}(e) \geq \max_{K \in \mathcal{K}} c_K(e)$$

□

Proof Given an edge $[m, n]$ in a minimal edge cover K, the upper bound cost function assumes the worst possible case. In particular, it assumes that, for each child m' of m, a cost of c_c and c_g, respectively, is incurred for all but one edges incident on m'; the remaining edge is assumed to incur a cost c_m for a move. (Recall that we assume that CPY and GLU both cost more than a MOV .) The only exception is when there is an edge $[m', n']$ for some child n' of n; such an edge clearly does not involve a move, and therefore contributes 0 units to the cost. An analogous worst-case scenario is assumed for each child n' of n. Furthermore, the cost of $[m, n]$ is highest when $|E(m)| = |E(n)| = 1$, which is what the upper bound function assumes, resulting in the overall upper bound.

Similarly, the lower bound function assumes the best possible case for each child m' of m. In particular, it assumes that no cost is incurred on behalf of m' except in those cases where matching m to n would force a child m' to be moved; in such a case, a cost contribution of c_m is added. Furthermore, note that the cost of an edge $[m, n]$ is lower as $E(m)$ and $E(n)$ are bigger. However, since K is restricted

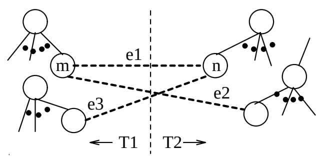


Figure 5.22: Applying pruning rules

to be a minimal edge cover, at least one of $E(m)$ and $E(n)$ must be a singleton set (containing just the edge $[m, n]$), or else there would be a path of length three in K, contradicting Lemma 5. Therefore, the cost of $[m, n]$ includes at least the lower of the two costs propagated from each of m, and n. Since this is precisely what the lower bound function defines c_{lb} to be, we see that the inequality for c_{lb} holds. $\square$

5.5.3 Pruning

We now use the upper and lower bound functions for the cost of an edge as defined above to introduce the pruning rules we use to reduce the size of the induced graph of the two trees being compared. Let $e_1 = [m, n]$ be any edge in the induced graph, as shown in Figure 5.22. Let e_2 be any edge incident on m, and let e_3 be any edge incident on n. Intuitively, our first pruning rules tries to remove edges with a lower bound cost that is so high that it is preferable to match each of its nodes using some other edges, given the existence of such edges with a suitably low upper bound cost.

Pruning Rule 1 Let $C_t = \max\{c_m, c_c, c_g\}$. If $c_{lb}(e_1) \geq c_{ub}(e_2) + c_{ub}(e_3) + 2C_t$ then prune e_1.

Example 5.5.1 To illustrate this rule, consider a tree T_1 containing, among others, two childless nodes 1 (label f) and 2 (label g). Similarly, T_2 contains childless nodes 3 (label g) and 4 (label f), among others. Say the costs c_m, c_c, and c_g are one unit each, while the update costs are $c_u(f, g) = 3$, and $c_u(f, f) = c_u(g, g) = 0$. Let us now consider if edge $e_1 = [1, 3]$ can be pruned because edges $e_2 = [1, 4]$ and $e_3 = [2, 3]$ exist. Since the nodes have no children, it is easy to compute $c_{lb}(e_1) = c_u(f, g) = 3$, $c_{ub}(e_2) = c_u(f, f) = 0$, and $c_{ub}(e_3) = c_u(g, g) = 0$. Since $C_t = 1$, we see that Pruning

Rule 1 holds and e_1 can be safely removed. The intuition is that in the worst case we can replace e_1 by edges e_2 and e_3. Using the latter edges could introduce at most the costs $c_{ub}(e_2)$ and $c_{ub}(e_3)$, plus the cost of two MOV, CPY, or GLU operations. The last factor can arise, for instance, if node 2 ends up being matched not only to node 3 but to another node in T_2. This means that node 2 needs to be copied, which would not have been necessary if we had kept edge e_1 and not used e_2. Similarly, the removal of edge e_1 may cause an extra glue operation for node 4. However, even in this worst case scenario, the costs would be less than the cost of updating the label of node 1 to that of node 2, so we can safely remove the $[1,2]$ edge.

$\square$

Our second pruning rule (already illustrated in Section 5.3) states that if it is less expensive to delete a node and insert another, we do not need to consider matching the two nodes to each other. More precisely, we state the following:

Pruning Rule 2 If $c_{lb}(e_1) \geq c_d(m) + c_i(n)$ then prune e_1.

Note that the above pruning rules are simpler to apply if we let e_2 and e_3 be the minimum-cost edge incident on m and n, respectively. The following lemma tells us that the pruning rules are conservative:

Lemma 8 *Let E_p be the set of edges pruned by repeated application of Pruning Rules 1 and 2. Let K_1 be any minimal edge cover of the graph B. There exists a minimal edge cover K_2 such that (1) $K_2 \cap E_p = \emptyset$, and (2) $C(K_2) \leq C(K_1)$.* $\square$

Proof The proof is by induction on the cardinality of E_p. When $|E_p| = 0$, the lemma is trivially true. Now assume that the lemma is true whenever $|E_p| \leq k$, for any $k \geq 0$. We will show that the lemma is also true when $|E_p| = k+1$. Each (successful) application of a pruning rule adds one edge to E_p. Consider the edge e_1 that was pruned last. Using the induction hypothesis for $E'_p = E_p - \{e_1\}$, we can generate an edge cover K'_1 such that (1) $K'_1 \cup E'_p = \emptyset$, and (2) $C(K'_1) \leq C(K_1)$.

If K'_1 does not contain e_1, let $K_2 = K'_1$. If K'_1 contains e_1, we modify K'_1 to obtain K_2 as follows. If e_1 was pruned using Pruning Rule 1, then let $K_2 = K'_1 - \{e_1\} \cup$

$\{e_2, e_3\}$, where e_2 and e_3 are the edges used in the application of Pruning Rule 1. Else, e_1 was pruned using Pruning Rule 2; in this case, let $K_2 = K_1' - \{e_1\} \cup \{[n_1, \ominus], [\oplus, n_2]\}$, where $e_1 = [n_1, n_2]$.

Clearly, $K_2 \cup E_p = \emptyset$. Since K_1' is an edge cover of B, and since the only nodes that could be possibly exposed as a result of removing e_1 from K_1' (namely, n_1 and n_2) are covered by the edges added to K_1' to obtain K_2, it follows that K_2 is also an edge cover of B. From the definition of the pruning rules, and Lemma 7 we see that $C(K_2) \leq C(K_1') \leq C(K_1)$. □

The pruning phase of our algorithm consists of repeatedly applying Pruning Rules 1 and 2. Note that the absence of edges raises the lower bound function, and lowers the upper bound function, thus possibly causing more edges to get pruned. Our algorithm updates the cost bounds for the edges affected by the pruning of an edge whenever the edge is pruned. By maintaining the appropriate data structures, such a cost-update step after an edge is pruned can be performed in $O(nlogn)$ time, where n is the number of nodes in the induced graph.

5.5.4 Computing a Min-Cost Edge Cover

After application of the pruning rules described above, we obtain a pruned induced graph, containing a (typically small) subset of the edges in the original induced graph. In favorable cases, the remaining edges contain only one minimal edge cover. However, typically, there may be several minimal edge covers possible for the pruned induced graph. We now describe how we select one of these minimal edge covers.

We first *approximate* the fair cost of every edge e that remains after pruning by its lower bound $e_{lb}(e)$. (We could have also use the upper bound, or an average of both bounds, since this is only an estimate.) Then, given these constant estimated costs, we compute a minimum-cost edge cover by reducing the edge cover problem to a bipartite weighted matching problem, as suggested in [PS82]. Since the weighted matching problem can be solved using standard techniques, we do not present the details in this chapter, noting only that given a bipartite graph with n nodes and e edges, the weighted matching problem can be solved in time $O(ne)$. For our application, e is the number of edges that remain in the induced graph after pruning.

5.6 Implementation and Performance

In this section, we describe our implementation of MH-DIFF, and discuss its analytical and empirical performance. Figure 5.23 depicts the overall architecture of our implementation, with rectangles representing the modules (numbered, for reference) of the program, and other shapes representing data. Given two trees T_1 and T_2 as input, Module 1 constructs the induced graph (Section 5.3.1). This induced graph is next pruned (Module 2) using the pruning rules of Section 5.5.3 to give the pruned induced graph. In Module 2, the update cost for each edge in the induced graph is computed using the domain-dependent comparison function for node labels (Section 5.2.2). The next three modules together compute a minimum-cost edge cover of the pruned induced graph using the reduction of the edge cover problem to a weighted matching problem [PS82]. That is, the pruned induced graph is first translated (by Module 3) into an instance of a weighted matching problem. This weighted matching problem is solved using a package (Module 4) [Rot] based on standard techniques [PS82]. The output of the weighted matching solver is a minimum-cost matching, which is translated by Module 5 into K_0, a minimum-cost edge cover of the pruned induced graph. Next, Module 6 uses the minimum-cost edge cover computed, to produce the desired edit script, using the method described in Section 5.4.2).

Recall that since we use a heuristic cost function to compute a minimum-cost edge cover, the edge cover produced by our program, and hence the edit script may not be the optimal one. We have also implemented a simple search module that starts with minimum-cost edge cover K_0 (see Figure 5.23) computed by our program and explores its neighborhood of minimal edge covers in an effort to find a better solution. The search proceeds by first exploring minimal edge covers that contain only one edge not in K_0. Next, we explore minimal edge covers containing two edges not in K_0, and so on. The intuition is that we expect the optimal solution to be "close" to the initial solution K_0. Although, in the worst case, such an exploration may be extremely time-consuming, note that as a result of pruning edges, the search space is typically

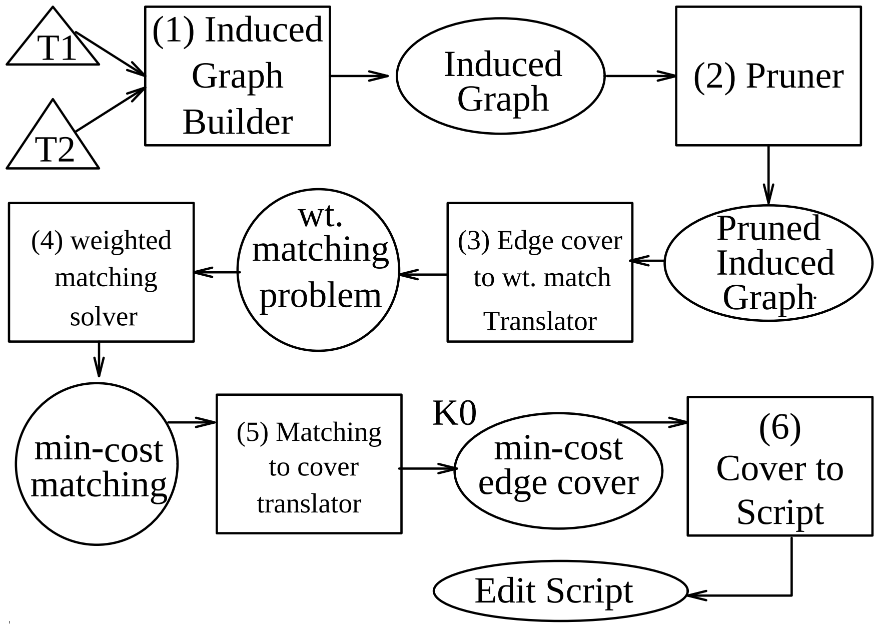


Figure 5.23: System Architecture

much smaller than the worst case.

We have used our implementation to compute the differences between query results as part of the Tsimmisand C^3projects at Stanford [CGMH+94, CGL+97]. These projects use the OEMdata model, which is a simple labeled-object model to represent tree-structured query results. In particular, we have run our system on the output of Tsimmisqueries over a bibliographic information source that contains information about database-related publications in a format similar to BibTeX. Since the data in this information source is mainly textual, we treat all labels as strings. For the domain-dependent label-update cost function, we use a weighted character-frequency histogram difference scheme that compares strings based on the number of occurrences of each character of the alphabet in them. For example, consider comparing the labels "foobar" and "crowbar." The character-frequency histograms are, respectively, $(a:1, b:1, f:1, o:2, r:1)$ and $(a:1, b:1, c:1, o:1, r:2, w:1)$. The difference histogram is $(c:-1, f:1, o:1, r:-1, w:-1)$. Adding up the magnitudes of the differences gives us 5, which we then normalize by the total number of characters in the strings (13), and scale by a parameter (currently 5), to get the update cost $(5/13) * 5 = 1.9$.

Let us now analyze the running time of our program. Let n be the total number of nodes in both input trees T_1 and T_2. Constructing the induced graph (Module 1, in Figure 5.23) involves building a complete bipartite graph with $O(n)$ nodes on each side. We also evaluate the domain-dependent label-comparison function for each pair of nodes, and store this cost on the corresponding edge. Thus, building the induced graph requires time $O(kn^2)$, where k is the cost of the domain-dependent comparison function. Next, consider the pruning phase (Module 2). By maintaining a priority queue (based on edge costs) of edges incident on each node of the induced graph, the test to determine whether an edge may be pruned can be performed in constant time. If the edge is pruned, removing it from the induced graph requires constant time, while removing it from the priority queues at each of its nodes requires $O(logn)$ time. When an edge $[m, n]$ is pruned, we also record the changes to the costs $c_{mc}(m, p(n))$, $c_{mc}(n, p(m))$, $c_{mf}(m, p(n))$, and $c_{mf}(n, p(m))$, which can be done in constant time. Thus, pruning an edge requires $O(logn)$ time. Since at most $O(n^2)$ are pruned, the total worst case cost of the pruning phase is $O(n^2logn)$. Let e be the number of edges that remain in the induced graph after pruning. The minimum-cost edge cover is computed in time $O(ne)$ by Modules 3, 4, and 5. The computation of the edit script from the minimum-cost edge cover can be done in $O(n)$ time by Module 6. (Note that the number of edges in a minimal edge cover is always $O(n)$.)

The number of edges that remain in the induced graph after pruning (denoted by e above) is an important metric for three main reasons. Firstly, as seen above, a lower number of edges results in faster execution of the minimum-cost edge cover algorithm. Secondly, a smaller number of edges decreases the possibility of finding a suboptimal edge cover, since there are fewer choices that need to be made by the algorithm. Thirdly, having a smaller number of edges in the induced graph reduces exponentially the size of the space of candidate minimal edge covers that the search module needs to explore.

Given the importance of the metric e, we have conducted a number of experiments to study the relationship between e and n. We start with four "input" trees representing actual results of varying sizes from our Tsimmissystem. For each input

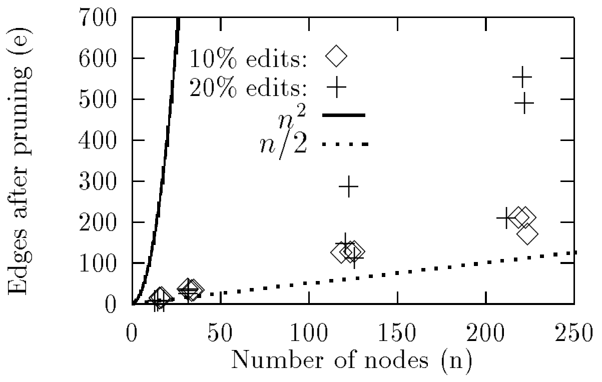


Figure 5.24: Effectiveness of pruning

tree, we generate a batch of "output" trees by applying a number of random edits. The number of random edits is either 10% or 20% of the number of nodes in the input tree. Then for each output tree, we run MH-DIFF on it and its original input tree. The results are summarized by the graph in Figure 5.24. The horizontal axis indicates the total number of nodes in the two trees being compared (and hence, in the induced graph). The vertical axis indicates the number of edges that remain after pruning the induced graph. Note that the ideal case (best possible pruning) corresponds to $e = \lceil n/2 \rceil$, since we need at least $\lceil n/2 \rceil$ edges to cover n nodes, whereas the worst case is $e = n^2$ (no pruning at all). For comparison, we have also plotted $e = n/2$ and $e = n^2$ on the graph in Figure 5.24. We observe that the relationship between e and n is close to linear, and that the observed values of e are much closer to $n/2$ than to n^2.

Note that in Figure 5.24 we have plotted the results for two different values of d, the percentage of random edit operations applied to the input tree. We see that, for a given value of n, a higher value of d results in a higher value of e, in general. We note that some points with a higher d value seem to have a lower value of e than the general trend. This is because applying d random edits is not the same as having the input and output trees separated by d edits, due to the possibility of redundant edit operations. Thus, some data points, even though they were obtained by applying d random edits, actually correspond to fewer changes in the tree.

We have also studied the quality of the initial solution produced by MH-DIFF. In particular, we are interested in finding out in what fraction of cases our method produces suboptimal initial solutions, and by how much the cost of the suboptimal solution exceeds that of the optimal. Given the exponential (in e) size of the search space of minimal edge covers of the induced graph, it is not feasible to try exhaustive searches on large datasets. However, we have exhaustively searched the space of minimal edge covers, and corresponding edit scripts, for smaller datasets. We ran 50 experiments, starting with an input tree T_1 derived as in the experiments for e above, and using 6 randomly generated edit operations to generate an output tree. We searched the space of minimal edge covers of the pruned induced graph exhaustively for these cases, and found that the MH-DIFF initial solution differed from the minimum-cost one in only 2 cases out of 50. That is, in 96% of the cases MH-DIFF found the minimum-cost edit script, and of course it did this in much less time than the exhaustive method. In the two cases where MH-DIFF missed, the resulting script cost about 15% more than the minimum cost possible.

5.7 Summary

In this chapter, we studied the problem of detecting changes from snapshots of structured or semistructured data that is represented using unordered trees. As in Chapter 4, we formalized this problem as the problem of computing a minimum-cost edit script that transforms one tree to another. However, the edit scripts studied in this chapter consist not only of operations that insert and delete nodes, update labels, and move subtrees, but also of operations that copy, and uncopy subtrees. Further, unlike the algorithms presented in Chapter 4, the work described in this chapter does not assume that the input trees have special properties such as layering. These changes to the problem definition, along with the fact that detecting changes in unordered trees is provably harder than the analogous problem of ordered trees, required us to rethink our strategy for solving the change detection problem.

Although the subtree operations of move, copy and uncopy are intuitive to use, they may often be interleaved in a complex manner to produce unintuitive results.

We illustrated such unintuitive edit scripts and described the difficulties they pose. In order to overcome these difficulties, we defined a structured edit script, in which the interleaving of different types of edit operations is restricted. We described how structured edit scripts allow us to benefit from the advantages of our expressive subtree operations while avoiding troublesome sequences of edit operations.

We defined the induced graph of two trees, and described the correspondence between minimal edge covers of this graph and structured edit scripts between the two trees. We presented an algorithm that uses this correspondence to compute a minimum-cost edit script. Since the problem is NP-hard, our algorithm uses heuristics to produce a good initial solution, followed by an optional search for better solutions. In practice, we have found that the initial solutions generated by our method are often optimal or close to optimal, allowing us to skip the search step. In Chapter 9, we study the performance of our method in more detail. In the next chapter, we study an alternate approach to avoiding the problems caused by arbitrary interleaving of subtree operations such as move, copy, and uncopy. In particular, we present a declarative specification of differences between trees that leads to simpler algorithms for change detection.

Chapter 6

Parallel Transformations

In Chapter 5 we presented techniques for computing differences between snapshots of data represented using unordered trees. We described the problems caused by edit scripts that combine edit operations on subtrees in a complex manner, yielding unintuitive results. In that chapter, we addressed these problems by defining structured edit scripts that restrict the interleaving of different types of edit operations. In this chapter, we explore an alternate method of overcoming the problem of unintuitive interleaving of edit operations: using a different model of tree transformation. Unlike the edit script model, which transforms trees procedurally by applying the operations in an edit script in sequence, this model transforms trees declaratively by functionally specifying the result of applying a transformation to a tree. Informally, we may think of this model as applying the edit operations in parallel. In addition to being more elegant than the restrictions required by structured edit scripts, this new model also results in simpler algorithms for finding a minimum-cost transformation between two trees. A slight drawback of this model is that it is often difficult for a person to understand the effect of a single edit operation independently of the other operations in the transformation. However, in many cases this drawback is not a significant problem and is outweighed by the advantages.

6.1 Introduction and Overview

As in Chapter 5, we study the problem of comparing *rooted, unordered, labeled trees*, such as those depicted in Figure 6.1. These trees may represent, for example, listings from a Web database containing information about movies. As in earlier chapters, tree nodes are represented by circles; each node has a label, indicated next to it, and an identifier, indicated inside the circle. In our Web example, the label of node 3 may represent a section heading, and its child nodes the paragraphs in the section. Recall from Chapters 4 and 5 that we use node identifiers for notational convenience only; we do not assume that these identifiers are object-identifiers or keys that can be used to match nodes in one tree with those in the other. (However, in cases where such object identifiers or keys exist, we can take advantage of them.)

Figure 6.1(a) also illustrates how changes to a tree T_1 can be represented by a *linear edit script*, such as those we studied in Chapters 4 and 5: $\mathcal{E}_1 = (cpy(5,2), cpy(4,6), mov(4,5))$. Recall that such a script is a sequence of *edit operations* that transforms T_1 into T_1'. For example, the first operation in our script, $cpy(5,2)$, makes a copy of the T_1 node 5, and places it under node 2. The new node has a new identifier, in this case 6. The edit operations commonly used in the literature are node *insertion*, node *deletion*, and node *relabeling*. Here, as in Chapter 5, we extend this set of edit operations by adding the subtree operations move, copy, and uncopy (or glue). As argued in Chapter 5, these operations allow us to express changes more succintly than is possible using only the three traditional operations. For example, when comparing structured documents, saying that a paragraph was moved is more helpful than saying that the sentences in the paragraph were deleted and then inserted somewhere else.

As in earlier chapters, our goal is to find a compact representation of the changes between two trees. If we use linear edit scripts, our goal is to find a "minimum-cost" script that transforms the first tree into one that is isomorphic to the second. We assign costs to operations and look for a minimum-cost script to ensure that the script does not do more work than needed. Unfortunately, this method of describing changes using linear edit scripts has several problems when used with subtree operations such

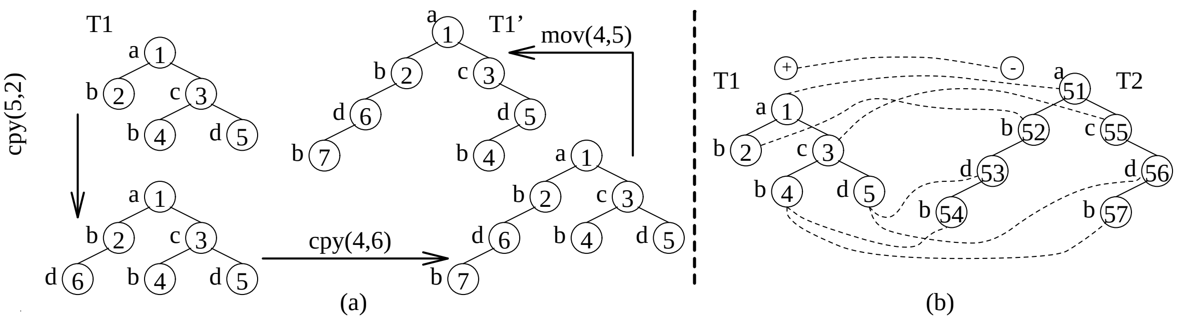


Figure 6.1: Applying a linear edit script.

as moves and copies.

The first problem is that it is difficult to understand an edit operation on its own, because its effect depends on the operations preceding it in the edit script. For example, consider the following edit script applied to T_1 in Figure 6.1(a): $\mathcal{E}_2 = (mov(4,5), cpy(5,2))$. If we focus on the copy operation, we would intuitively expect it to produce a copy of node 5, with node 2 as the parent. However, because of the preceding move, the copy actually produces a copy of both nodes 4 and 5. In fact, $\mathcal{E}_2$ has the same effect as the script $\mathcal{E}_1$ discussed earlier. This equivalence is not clear from the edit scripts themselves; we need to actually apply the edit scripts to discover it.

Another problem with the linear edit script model is that it may result in very unintuitive edit scripts. For example, consider structured documents, and suppose the cost of a copy operation is 5 units, while the cost of a move operation is 1 unit. Consider now a script that moves the subtree rooted at a node n_1 to below another node n_2, copies the subtree rooted at n_2 (thus also making a copy of the n_1 subtree), and then moves both the original and the copy of the subtree rooted at n_1 to other locations. We observe that the sole purpose of the initial move operation is to get a "free" copy of the subtree at n_1, thus reducing the overall cost of the edit script. However, this "trick" is not very intuitive in the application context: If n_1 and n_2 represent paragraphs, the above script says that paragraph n_1 is temporarily moved under n_2, not because n_1 is at all related to n_2, but simply to make it cheaper to make the copies we eventually need of n_1 that will go elsewhere.

In this chapter we present a novel method to represent changes and to compare trees that avoids these problems. The intuitive idea is to apply edit operations "in parallel" as opposed to in sequence. That is, we apply a *set* of edit operations, called a *transformation*, to a tree by first disassembling the given tree into "*chunks*," then operating on each chunk independently, and finally reassembling the resulting chunks to get the final tree. (In Section 6.2 we describe our model in detail.) Our model is free from the the unintuitive artifacts resulting from the interdependencies between edit operations in the linear edit script model. Even more importantly, searching for a minimum-cost parallel transformation is simpler than searching for a minimum-cost edit script (when moves and copies are allowed). This simplicity is because the essential information in a transformation, including its cost, can be compactly represented in a *signature*. Thus, we can search for a minimum-cost signature and then map it back to the corresponding transformation. In this chapter we show how signatures are constructed, and how they map to transformations. The mapping between signatures and transformation is independent of the cost model used, making our methods for detecting changes useful in diverse application domains.

The idea of working with signatures is widely used in the literature of differencing algorithms, in various forms (such as "traces" or matchings) [WF74, Mye86, ZS89, Yan91]. However, the introduction of move and copy operations makes it hard to recover a script from a signature, and this makes it difficult to detect changes using signatures. To illustrate some of these difficulties, Figure 6.1(b) shows the "traditional" signature of the edit script in Figure 6.1(a). The trees T_1 and T_2 represent the initial and final trees (respectively) from Figure 6.1(a). However, note that node identifiers in T_2 are different from those in T_1' (and T_1) because we do not know yet how T_2 was obtained from T_1. Intuitively, the signature is a relation (dashed lines) that maps each node in T_2 to the node or nodes in T_1 from which it is "derived." For example, if nodes $n_1, n_2 \in T_2$ are copies of a common node $m \in T_1$, the signature maps m to both n_1 and n_2. Deleted T_1 nodes are mapped to a special node $\ominus$, while inserted T_2 nodes are mapped to a special node $\oplus$. In our sample script, there were no inserts or deletes, so the signature just links $\ominus$ to $\oplus$. (More formally, the signature is a minimal edge cover of the complete T_1, T_2 bipartite graph; see Section 6.3.)

If our search for a minimum-cost signature yields the signature of Figure 6.1(b), it is hard to recover the corresponding minimum-cost edit script. In particular, we note that since there are two dashed edges incident on nodes 4 and 5, we may conclude that these nodes were copied by the edit script. Similarly, since node 4 does not have a "partner" (by dashed edges) whose parent matches the parent of node 4, we may conclude that node 4 is moved. With some bookkeeping, this reasoning recovers the original edit script $\mathcal{E}_1 = (cpy(5,2), cpy(4,6), mov(4,5))$. Unfortunately, this edit script is not a minimum-cost edit script (assuming, say, unit costs for edit operations); the edit script $\mathcal{E}_2 = (mov(4,5), cpy(5,2))$ achieves the same result with one fewer operation. By moving node 4 to under node 5 before node 5 is copied, we get a "free" copy of node 4. Thus, to recover the minimum-cost edit script from the signature we would need to consider all such possibilities of saving operations by "piggy-backing" them on others. As we will see, our parallel transformation model does not have these problems: it is easy to recover a minimum-cost transformation from a signature, making the search for a minimum-cost transformation efficient and simple.

In summary, our main contributions in this chapter are the following:

- We present a novel model for tree transformations that permits expressive operations such as subtree move and copy and avoids the problems caused by arbitrary interleaving of such operations in a linear edit script model.

- We describe how the essential features of transformations in this model are captured using representative signatures, and describe how these signatures simplify algorithms for finding a minimum-cost transformation between two trees.

- We present algorithms for mapping transformations to signatures and vice-versa, and describe techniques for computing signatures that produce good transformations.

The rest of this chapter is organized as follows. We first define our model of tree transformations in Section 6.2 below. In Section 6.3, we define the signature of a transformation, and formalize the manner in which it succinctly captures the essence of a transformation by proving our main results of this chapter, Theorems 5 and 6. Section 6.4 presents the application of these ideas by describing methods for computing a desirable signature for two given trees, and Section 6.5 summarizes this chapter.

6.2 Transformation Model

Let $\mathcal{N}$ be a domain of node identifiers, and let $\mathcal{L}$ be a domain of labels. A *rooted, unordered, labeled tree* T is a 4-tuple (N, r, p, l), where $N \subset \mathcal{N}$ is called the set of nodes in T, $r \in N$ is a distinguished node, called the *root* of T, $p : N - \{r\} \to N$ is a cycle-free function called the *parent* function of T, and $l : N \to \mathcal{L}$ is called the *label* function of T. (By cycle-free, we mean $p^k(n) \neq n$ for any $n \in N$ and $k > 0$.) Henceforth in this chapter, by trees we mean rooted, unordered, labeled trees.

In our model, a *transformation* is a set of edit operations (defined below). Each edit operation in a transformation has a unique identifier. In what follows, we often need a way to refer to nodes produced by insertion or copy operations. (For example, we may wish to update a node produced by a copy operation.) We use *node handles* for this purpose. In particular, we use the following notation for node handles: `f(`n`,`i`)`, where $n \in \mathcal{N}$ and $i \in Z^+$, refers to the copy of node n produced by the copy operation (with identifier) i; `f(0,`i`)` refers to the node produced by insertion operation i; `f(`n`,0)` refers to the node n. We denote the set of all node handles by $\mathcal{H}$. (Here and in the rest of this chapter, we use `type font` to represent literal strings, and *italics* to represent non-literals.) The *edit operations* on trees are introduced below, along with an informal description of their effect; the formal definition follows as Definition 6.2.3.

Delete: `del(`h`, `j`)`, where $h =$ `f(`n`,0)` for $n \in \mathcal{N}$. Intuitively, this edit operation deletes the node n. (As we shall see later in Definition 6.2.2, the children of n are either deleted, moved, or glued.) In this and the following edit operations, the last argument $j \in Z^+$ is a unique identifier of the edit operation; for brevity, j is often omitted when not needed.

Insert: `ins(`h`,` l`,` j`)`, where $h \in \mathcal{H}$, $l \in \mathcal{L}$, and $j \in Z^+$. Intuitively, this edit operation inserts a node with parent (the node corresponding to) h and label l. (The newly created node has handle `f(0,`j`)`.)

Update: `upd(`h`,` l`,` j`)`, where $h \in \mathcal{H}$, $l \in \mathcal{L}$, and $j \in Z^+$. Intuitively, this edit operation changes the label of node h to l.

Move: `mov(`h_1`,` h_2`,` j`)`, where $h_1 =$ `f(`n`,0)` for $n \in \mathcal{N}$, $h_2 \in \mathcal{H}$, and $j \in Z^+$. Intuitively, this edit operation moves the chunk (defined below) rooted at n, making h_2 its new parent.

Copy: `cpy(`h_1`,` h_2`,` j`)`, where $h_1 =$ `f(`n`,0)` for $n \in \mathcal{N}$, $h_2 \in \mathcal{H}$, and $j \in Z^+$. Intuitively, this edit operation copies the chunk rooted at n, making h_2 the parent of the copy.

Glue: `glu(`h_1`,` h_2`,` j`)`, where $h_1 =$ `f(`n_1`,0)` for $n_1 \in \mathcal{N}$, $h =$ `f(`n_2`,0)` for $n_2 \in \mathcal{N}$, and $j \in Z^+$. Intuitively, glue is the inverse of a copy operation; it causes the chunk rooted at n_1 to disappear by "gluing" it over the chunk rooted at n_2.

As noted in Section 6.1, the first step to applying a transformation is the disassembly of the given tree into "chunks." Chunks, or *disassembly components* (see below), are produced by breaking up the tree at every node that is "operated on" by an edit operation. The break up points are called *disassembly points*.

Definition 6.2.1 Given a tree $T = (N, r, p, l)$ and a transformation F, we define the set of *disassembly points*, $dp(T, F)$, as follows:

$$\begin{aligned} dp(T,F) \quad = \quad & \{n \in N \mid n = r \vee \exists\, del(f(n,0)) \in F \vee \\ & \exists\, mov(f(n,0),h) \in F \vee \exists\, cpy(f(n,0),h) \in F \vee \\ & \exists\, glu(f(n,0),h) \in F \vee \exists\, glu(h, f(n,0)) \in F\} \end{aligned}$$

With reference to a transformation F applied to a tree T, we define the *nearest disassembly ancestor* $nda(n, T, F)$ of a node $n \in N$ to be the nearest (not necessarily proper) ancestor of n that belongs to $dp(T, F)$. Further, the *disassembly component* ("chunk") of n is defined as $dc(n, T, F) = \{n' \in N \mid nda(n') = nda(n)\}$. □

When discussing a given tree and transformation, we abbreviate $nda(n, T, F)$ by $nda(n)$, and $dc(n, T, F)$ by $dc(n)$. Not every transformation as defined above can be applied to a given tree. Given a tree T and a transformation F, we define the notion of *validity* of F over T as follows.

Definition 6.2.2 A transformation F is said to be *valid* for a tree $T = (N, r, p, l)$ if the following conditions hold.

1. The transformation F is *well-formed*; that is, the following hold:
   (a) Identifiers of edit operations in F are unique.
   (b) For each node handle $f(n, 0)$ appearing in F: $n \in N$.
   (c) For each $f(0, i)$ in F: $ins(h, l, i) \in F$ for some $h \in \mathcal{H}$ and $l \in \mathcal{L}$.
   (d) For each $f(n, i)$ in F with $i > 0$: $n \in N$, and $cpy(f(nda(n), 0), h, i) \in F$ for some $h \in \mathcal{H}$.
   (e) If $mov(f(n_1, 0), f(n_2, 0)) \in F$, then n_1 is not an ancestor of n_2 in T.
   (f) For each $glu(f(n_1, 0), f(n_2, 0), i) \in F$, there exists an isomorphism g_i between $dc(n_1)$ and $dc(n_2)$. More precisely, there exists a function $g_i : dc(n_1) \to dc(n_2)$ that is one-to-one, onto, preserves the parent function p, and "preserves labels" in the sense that $g_i(x) = y$ implies the following: If $upd(x, l) \in F$ then either $l(y) = l$ or $upd(y, l) \in F$; else either $l(y) = l(x)$ or $upd(y, l(x)) \in F$.
2. For each node n in T, at most one of the following types of edit operations is in F: $del(f(n, 0))$, $cpy(f(n, 0), n')$, $glu(f(n, 0), h)$, and $glu(h, f(n, 0))$. Further, for each $n \in T$, there is at most one operation of the form $del(f(n, 0))$, at most one operation of the form $mov(f(n, 0), h)$, and at most one operation of the form $glu(f(n, 0), h)$ in F. Finally, no node is updated more than once.
3. If F contains $del(f(nda(p(n)), 0))$ or $glu(f(nda(p(n)), 0), h)$ (for some n), then one of $del(f(n, 0))$, $mov(f(n, 0), h)$, and $glu(f(n, 0), h)$ is in F.

□

The last condition in the above definition is not strictly necessary, but is used to make deletes (respectively, glues) more symmetrical to inserts (respectively, copies). Since any children of inserted (copied) nodes need to be inserted, moved, or copied to that location, we require that any children of a deleted (respectively, glued) node be deleted, moved, or glued.

For ease of explanation, we henceforth assume, without loss of generality, that no edit operation acts on the root of a tree. (We can always add an artificial root to any tree to ensure this property holds.) We are now ready to define the tree $F(T)$ obtained by applying a transformation F to a tree T. Intuitively, we start with a working copy T' of T, and break T' into chunks (i.e., the disassembly components defined above). Nodes deleted by F are removed. Next, copy, update, glue, and move operations in F are applied to the chunks of T'. Nodes corresponding to insertion operations in F are created. Finally, the chunks are reassembled to yield the tree $F(T)$. Formally, we define $F(T)$ as follows:

Definition 6.2.3 Given a tree $T = (N, r, p, l)$ and a transformation F valid for T, the result of *applying the transformation* F to T is a tree $F(T) = (N', r', p', l')$ where N', p', r', and l' are defined below. In the following, we use a skolem function $f' : \mathcal{N} \times Z \to \mathcal{N}$ such that $f'(n, i)$ intuitively represents the node in T' referenced by the node handle `f(`n`, `i`)` in F.

$$\begin{aligned}
N' &= \{f'(n,0) \mid n \in N,\ del(f(n,0)) \notin F,\ glu(f(nda(n),0),h) \notin F\} \\
&\cup \{f'(0,i) \mid ins(h,l,i) \in F\} \\
&\cup \{f'(n,i) \mid cpy(f(nda(n),0),h,i) \in F\} \\
r' &= f'(r,0) \\
p'(f'(0,i)) &= f'(n_2,j),\ \text{where } ins(f(n_2,j),l,i) \in F \\
p'(f'(n,0)) &= f'(n_2,i),\ \text{if } mov(f(n,0),f(n_2,i)) \in F \\
& f'(p(n),0), \text{otherwise} \\
p'(f'(n,i)) &= f'(n_2,j),\ \text{if } cpy(f(n,0),f(n_2,j),i) \in F \\
& f'(p(n),i),\ \text{otherwise}
\end{aligned}$$

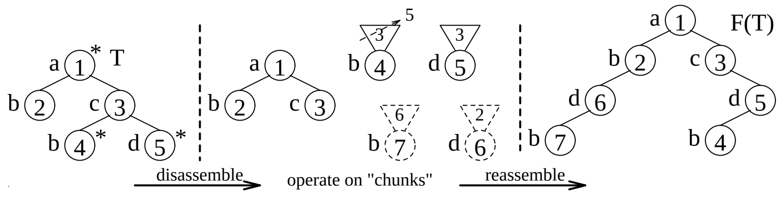


Figure 6.2: Applying the transformation in Example 6.2.1

$$\begin{aligned} l'(f'(0,i)) &= l_1, \text{ where } ins(h,l_1,i) \in F \\ l'(f'(n,i)) &= l_1, \text{ if } upd(f(n,i),l_1) \in F \\ & \quad l(n), \text{ otherwise} \end{aligned}$$

□

Example 6.2.1 Consider the tree T depicted in Figure 6.2, and the following transformation F: $\{mov(f(4,0), f(5,0)), cpy(f(4,0), f(5,101)), cpy(f(5,0), f(2,0), 101)\}$. The disassembly points of T by F are marked by an asterisk; they are, intuitively, the nodes in T that are acted on by edit operations in F (in addition to the root). The tree T's disassembly components also shown in the figure; the stubs on the nodes indicate the parent of the chunk. The results of applying the operations in F to the chunks are indicated using dashed lines. In particular, the operation $mov(f(4,0), f(5,0))$ results in the parent of the chunk rooted at node 4 to change from node 3 to node 5. The operation $cpy(f(5,0), f(2,0), 101)$ results in the duplication of the chunk rooted at node 5, producing a new node with identifier 6. (Thus, by our node handle notation, $6 = f(5,101)$.) Similarly, operation $cpy(f(4,0), f(5,101))$ results in a copy of the chunk rooted at node 4. Note that the parent of the copy is node 6 because $f(5,101) = 6$. Finally, the tree $F(T)$ obtained by reassembling the chunks (using the stubs) is also shown.

Note that the result of applying transformation F to tree T is independent of the order in which the edit operations in F are applied. For example, if we had considered applying the move operation after both copy operations, the result would be the same as above. Thus we can intuitively understand the effect of each edit operation on the chunks without worrying about the actions of other edit operations in the transformation. For example, we do not have to worry about a copy operation acting on a chunk resulting in surreptitious copies of other chunks (due to those chunks first being moved to below the copied chunk), as is the case when using the linear edit scripts described in Section 6.1. □

Recall (from Section 6.1) that we are interested in finding a minimum-cost transformation between two given trees. We define the *cost of a transformation* to be the sum of the costs of its constituent edit operations. The cost of each edit operation is given by some application dependent function. For example, in an application comparing structured documents, the cost of updating (tree nodes representing) words would depend on how similar the old and new values are. Thus updating "cat" to "cats" may cost 0.1 unit, while updating "cat" to "dinosaur" may cost 2 units. We do not discuss details of the cost model in this work, since our main results do not depend on them.

6.3 Representative Signatures of Transformations

In Section 6.1 we introduced signatures as a concise representation of the essential information in a transformation. In particular, given a signature S of a transformation F, one can easily recover a transformation F' that is essentially identical to F. Signatures satisfying this property are called *representative signatures*, and they are useful tools for computing a minimum-cost transformation. (Searching in signature space is more convenient than searching in the space of all possible transformations.) Below, we first define the signature $S(F, T)$ of a transformation F applied to a tree T. We then describe how to recover from $S(F, T)$ a transformation F' that is essentially identical to F (as indicated by Theorems 5 and 6), thus showing that our signatures are representative.

Intuitively, we may think of generating signatures by using the following procedure: We start with the given tree T and a tree T' that is isomorphic to T. We create signature edges (as distinguished from tree edges) connecting each node in T to its partner in T' (based on the isomorphism). We apply the transformation F to T', updating our set of signature edges in the process as follows: When a node is deleted, signature edges incident on it are redirected to $\ominus$; similarly, we introduce signature edges connecting inserted nodes to $\oplus$. When a subtree is copied, we connect the copy c of a node n to all the nodes to which n is connected; glues are handled analogously. Moves and updates do not affect the set of signature edges. We are then left with a set of signature edges connecting nodes in T to nodes in the transformed T'. Formally, we have the following definition for signatures:

Definition 6.3.1 Let F be a transformation that is valid for a tree $T = (N, r, p, l)$, and let $F(T) = (N', r', p', l')$.. We define the *signature* of F on T to be a relation $S(F, T) \subset (N \cup \{\oplus\}) \times (N' \cup \{\ominus\})$ as follows. The function f' is from Definition 6.2.3, the function g_j is from Definition 6.2.2, and $\oplus$ and $\ominus$ are distinguished reserved nodes in $\mathcal{N}$.

$$\begin{aligned}
S(F,T) &= \{(\oplus, f'(0,i)) \mid ins(h,l,i) \in F\} \\
&\cup \{(n, \ominus) \mid del(f(n,0)) \in F\} \\
&\cup \{(n, f'(n,i)) \mid n \in F,\ cpy(f(nda(n),0),h,i) \in F\} \\
&\cup \{(n, f'(n',0)) \mid n \in F,\ glu(f(nda(n),0),h,j) \in F,\ g_j(n,n')\} \\
&\cup \{(n, f'(n,0)) \mid n \in F,\ del(f(n,0)) \notin F,\ glu(f(nda(n),0),h) \notin F\} \\
&\cup \{(\oplus, \ominus) \mid \nexists ins(\ldots) \in F \ \vee\ \nexists del(\ldots) \in F\}
\end{aligned}$$

□

We define the *induced graph* of two trees $T_1 = (N_1, r_1, p_1, l_1)$ and $T_2 = (N_2, r_2, p_2, l_2)$ to be the complete bipartite graph $IG(T_1, T_2) = (U, V, U \times V)$, where $U = N_1 \cup \{\oplus\}$ and $V = N_2 \cup \{\ominus\}$. In general, signatures are edge covers of the induced graph. However, we will now show that we can restrict our attention to minimal edge covers, defined below:

Definition 6.3.2 Given a bipartite graph $B = (U, V, E)$, with distinguished nodes $\oplus \in U$ and $\ominus \in V$, a set $K \subseteq E$ is called an *edge cover* of B if each node in $U \cup V$ is incident on at least one edge in K. The set K is said to be a *minimal edge cover* if it is an edge cover that (1) does not contain any paths of length three, and (2) does not contain any paths of length two ending at $\oplus$ or $\ominus$. □

Note that above definition implies that no proper subset of a minimal edge cover is an edge cover. The following lemma shows that for any tree T and valid transformation F, $S(F,T)$ is a minimal edge cover of $IG(T, F(T))$.

Lemma 9 *For any transformation F valid for a tree T, $S(F,T)$ is a minimal edge cover of $IG(T, F(T))$.* □

Proof Let us first show that each node in $IG(T, F(T))$ is incident on at least one edge in $S(F,T)$. If there is an insert operation in F, then $[\oplus, f'(0,i)] \in S(F,T)$; if not, $[\oplus, \ominus] \in S(F,T)$. Thus, $\oplus$ is covered by $S(F,T)$. An analogous argument holds for $\ominus$. Now consider any node $n \in T$. If $del(f(n,0)) \in F$, then $[n, \ominus] \in S(F,T)$; else if $glu(f(n,0), f(n',0)) \in F$, then $[n, f'(n',0)] \in S(F,T)$; else $[n, f'(n,0)] \in S(F,T)$. Thus, in all cases n is covered by $S(F,T)$. Now consider a node $n' \in T'$. If $n' = f'(0,i)$, then $ins(h,l,i) \in F$, implying $[\oplus, n'] \in S(F,T)$; else if $n' = f'(n,0)$, then (from Definition 6.2.3) $del(f(n,0)) \notin F$ and $glu(f(n,0), h) \notin F$ implying $[n,n'] \in S(F,T)$; else $n' = f'(n,i)$ for $i > 0$, implying $cpy(f(n,0), h, i) \in F$ so that $[n,n'] \in S(F,T)$. Thus, in all cases n' is covered by $S(F,T)$. We have thus shown that each node in $IG(T, F(T))$ is covered by $S(F,T)$.

Let us now show that $S(F,T)$ is a *minimal* edge cover of $IG(T, F(T))$. We first show that there is no path of length two ending at $\oplus$ or $\ominus$. Consider an edge $[\oplus, n']$. From Definition 6.3.1, it follows that $n' = f'(0,i)$, for some edit operation identifier i. Using the uniqueness of edit operation identifiers, we see that there can be no other edge incident such a node n'. Thus there are no paths of length two terminating at $\oplus$. An analogous argument shows that there are no paths of length two terminating at $\ominus$.

We now show that there are no paths of length three in $S(F,T)$. Let, if possible, n_1, n_2, n_3, n_4 be a path of length three in $S(F,T)$ such that $n_1 \in T$, implying $n_2 \in F(T)$, $n_3 \in T$, and $n_4 \in F(T)$. Since we have shown that there are no paths of length two incident on $\oplus$ or $\ominus$, it follows that $n_i \neq \oplus, \ominus$, for $i = 1 \ldots 4$. From Definition 6.3.1, we see that if $n \in T$ is a node with multiple edges in $S(F,T)$ incident on it, then $nda(n)$ is acted on by a copy operation in F. Now n_3 is such a node, implying $cpy(nda(n_3),x) \in F$. We also observe that if $n' \in F(T)$ is a node with multiple edges $\{[m_i,n']\}$ $(i = 1 \ldots k,\ k > 1)$ in $S(F,T)$ incident on it, then $n' = f'(m_{i^*},0)$, where $i^* \in [1,k]$, and $glu(nda(m_i), nda(m_{i^*})) \in F$ for all $m_i \neq m_{i^*}$; thus there is a glue operation acting on each $nda(m_i)$, $i \in 1 \ldots k$. Now n_2 is such a node, with edges $[n_1,n_2]$ and $[n_3,n_2]$ incident on it. Therefore, there is a glue operation acting on $nda(n_3)$. Thus $nda(n_3)$ is acted on by both a copy and a glue operation, contradicting the validity of F (Definition 6.2.2). We therefore conclude that no such path exists in $S(F,T)$, proving minimality.

□

Later we show that the converse of the above lemma is also true; that is, for every minimal edge cover K of $IG(T_1,T_2)$ there exists some transformation F' such that $F'(T_1) = T_2$, and $S(F',T_1) = K$. Therefore, when searching for the signature of a minimum-cost transformation between two given trees, it suffices to search over the space of all minimal edge covers of their induced graph. Once we have found a minimal edge cover that is the signature of a minimum-cost transformation, we can easily recover the actual transformation from it, as described below.

To recover a transformation from a minimal edge cover K of the induced graph of two trees T_1 and T_2, we proceed in two steps. The first step consists of determining the disassembly points of the required transformation. In the second step, we use these disassembly points to generate the actual edit operations in the transformation. Intuitively, we determine the disassembly points of T_1 and T_2 using K as follows. First, the tree roots, and the special nodes $\oplus$ and $\ominus$ are deemed disassembly points. Next, any node whose partners (by K) are in any way "different" from the partners of its parent is a disassembly point. We say the partners of a node n are "different" from those of its parent $p(n)$ if there is some partner of n whose parent is not a partner of $p(n)$, or vice versa. Finally, any partner of a disassembly point is also a disassembly point. Definition 6.3.3 below presents the formal definition of the *cover disassembly points* of trees T_1 and T_2 by an edge cover K of their induced graph; we denote this set of points by $cdp(T_1,T_2,K)$.

Definition 6.3.3 Let $T_1 = (N_1, r_1, p_1, l_1)$ and $T_2 = (N_2, r_2, p_2, l_2)$ be two trees and let K be a minimal edge cover of their induced graph $B = IG(T_1, T_2)$. We define the *cover disassembly points* of T_1 and K as the following set: $cdp(T_1, K) \subset N$.

$$\begin{array}{lcl}
cdp(T_1, K) & = & \{r\} \\
& \cup & \{m \in N \mid [m, \ominus] \in K\} \\
& \cup & \{m \in N \mid \exists [m, n] \in K : [p(m), p(n)] \notin K\} \\
& \cup & \{m \in N \mid \exists [m, n] \in K : (\exists [m', n] \in K : [p(m'), p(n)] \notin K)\} \\
& \cup & \{m \in N \mid \exists [m, n_1], [m, n_2] \in K : p(n_1) = p(n_2)\} \\
& \cup & \{m \in N \mid \exists [m, n] \in K : (\exists [m_1, n], [m_2, n] \in K : p(m_1) = p(m_2))\} \\
& \cup & \{m \in N \mid \exists [p(m), n] \in K : (\not\exists [m, n'] \in K : p(n') = n)\} \\
& \cup & \{m \in N \mid \exists [m, n] \in K : ([m', n] \in K : (\not\exists [m'', n] \in K : p(m'') = m))\}
\end{array}$$

We define the cover disassembly points of T_2 and K as $cdp(T_2, K) = \{n \in T_2 \mid [m, n] \in K,\ m \in cdp(T_1, K)\} \cup \{n \in N' \mid [\oplus, n] \in K\}$. We also define the *cover nearest disassembly ancestor*, $cnda(n)$, and the *cover disassembly component*, $cdc(n)$, of a node $n \in T_1 \cup T_2$ analogously to the corresponding definitions in Definition 6.2.1. □

The following lemma shows that the set of points given by this definition is exactly the set of disassembly points of a minimum-cost transformation whose signature is the given edge cover.

Lemma 10 *Let $T = (N, r, p, l)$ and $T' = (N', r', p', l')$ be two trees, and let K be a minimal edge cover of their induced graph $IG(T, T')$.*

$$cdp(T, K) = dp(T, F(K, T, T'))$$

□

Proof Let us first show that $cdp(T, K) \subseteq dp(T, F(K, T, T'))$. Let m be any node in $cdp(T, K)$. If $m = r$, then $m \in dp(T, F(K, T, T'))$ since the root is always included in $dp(T, F(K, T, T'))$ (Definition 6.2.1). If $[m, \ominus] \in K$, then $del(m) \in F(K, T, T')$

(by Definition 6.3.6), implying $m \in dp(T, F(K, T, T'))$. Now for all $m \in cdp(T, K)$ other than those considered above, Definition 6.3.6 generates either a *mov*, *cpy*, or *glu* operation, implying $m \in dp(T, F(K, T, T'))$. Thus $cdp(T, K) \subseteq dp(T, F(K, T, T'))$.

Let us now show that $dp(T, F(K, T, T')) \subseteq cdp(T, K)$. Let m be any node in $dp(T, F(K, T, T'))$. If $m = r$, $r \in cdp(T, K)$ as required. If $del(m) \in F(K, T, T')$, Definition 6.3.1 implies $[m, \ominus] \in K$, implying $m \in cdp(T, K)$. Otherwise, either $mov(m, h)$, $cpy(m, h)$, $glu(h, m)$, or $glu(m, h)$ is in $F(K, T, T')$, implying m is in $cdp(T, K)$. Thus $dp(T, F(K, T, T')) \subseteq cdp(T, K)$ which, with our earlier result, completes the proof. □

Once we have determined the disassembly points as described above, we recover the actual transformation as follows: Nodes matched to the special node $\ominus$ are deleted. Nodes matched to $\oplus$ indicate nodes to be inserted. A one-to-one (edge cover) edge incident on a disassembly point signifies a move operation. For a disassembly point on which $k > 1$ (edge cover) edges are incident, we generate $k-1$ copy operations and zero or one move operation. The edge for which a copy operation is not generated is called "distinguished." The choice of this distinguished edge is significant only when we can avoid a move operation by choosing as distinguished edge an edge that connects two nodes whose parents also "match." This intuition is formalized by the following definition:

Definition 6.3.4 Let $T = (N, r, p, l)$ and $T' = (N', r', p', l')$ be two trees, and let K be a minimal edge cover of their induced graph $IG(T, T')$. Without loss of generality, let $[r, r']$ be the only edge in K incident on either of r or r'. Let $E(n)$ denote the edges in K that are incident on a node $n \in N \cup N'$. We define the *matched edge set* of a node $n \in N \cup N' - \{r, r'\}$ such that $[\oplus, n], [n, \ominus] \notin K$ as the set $E'(n)$ below:

$$\begin{aligned} E'(n) &= \{[n, n'] \in E(n) \mid [p(n), p(n')] \in K\}, \text{ if } n \in N \\ & \{[n', n] \in E(n) \mid [p(n'), p(n)] \in K\}, \text{ otherwise} \end{aligned}$$

Further, let us define the *distinguished edge* $de(n)$ incident on any $n \in cdp(T, K)$ as follows:

$$de(n) = \begin{cases} e, \text{ if } E(n) = \{e\} \\ \text{an arbitrary edge in } E'(n), \text{ if } |E(n)| > 1,\ E'(n) \neq \emptyset \\ \text{an arbitrary edge in } E(n), \text{ otherwise} \end{cases}$$

Note that our definition of de implies $de(r) = de(r') = [r, r']$, $de(m) = [m, \ominus]$ for all $[m, \ominus] \in K$, and $de(n) = [\oplus, n]$ for all $[\oplus, n] \in K$. Furthermore, we extend the definition of de to all nodes in $N \cup N'$ by defining $de(n)$ for $n \in N \cup N' - cdp(T, K)$ as follows:

$$de(n) = [m, n], \text{ where } de(cnda(n)) = de(cnda(m))$$

(From the definition of cdp and $cnda$, it is easy to observe that for any $n \in N \cup N' - cdp(T, K)$, there is exactly one node m such that $de(cnda(n)) = de(cnda(m))$.) □

Given two trees T_1 and T_2, and a minimal edge cover K of their induced graph, Definition 6.3.6 below presents the details of recovering a transformation F from K based on the intuition described above. Recall that we require that $T_1' = F(T_1)$ be isomorphic to T_2. When generating the edit operations in F, we often need to refer to the node in T_1' that corresponds (by the isomorphism) to a certain node in T_2. As described in Section 6.2, nodes are referenced in edit operations using node handles; i.e., expressions of the form `f(n,i)`. Thus, we need a way to map each node in $n \in T_2$ to a node handle that represents its partner in T_1'; we call such a node handle the *representative handle* of n, and denote it by $h(n)$. Using the definition of node handles in Section 6.2, it is easy to observe that the representative handle of a node n in T_2 that is matched to $\oplus$ is `f(0,` i`)`, where i is the identifier of the insertion operation that produces the node in T_1' that is isomorphic to n. Further, for a node $n \in T_2$ that is matched to exactly one node $m \in T_1$ by a one-to-one edge, we have $h(n) =$ `f(`m`,0)`. The case in which n is matched to more than one node in T_1 is similar; we simply pick the node m such that $[m, n]$ is the distinguished edge incident on n. Finally, if a node $n \in T_2$ is matched to a node $m \in T_1$ that has more than one edge incident on it, we have two cases: If $[m, n]$ is the distinguished edge incident on m, it means that n is the node corresponding to m, and $h(n) = \texttt{f(}m\texttt{,0);}$ otherwise n represents a copy of m, and $h(n) = \texttt{f(}m\texttt{,}i\texttt{)}$, where i is the identifier of the copy operation that produces the node in T_1' that is isomorphic to n. The following definition formalizes the above intuition.

Definition 6.3.5 Let $T = (N, r, p, l)$ and $T' = (N', r', p', l')$ be two trees, and let K be a minimal edge cover of their induced graph $B = IG(T, T') = (U, V, E)$. Without loss of generality, let $[r, r']$ be the only edge in K incident on either of r or r'. We define the *representative handle* $h(n)$ of a node $n \in T'$ as follows, where we use `type font` to represent literal strings, and *italic font* to represent non-literals, and where σ is a function that maps edges to arbitrary, unique, positive integers:

$$\begin{aligned} h(n) &= \texttt{f(0,}\sigma(de(n))\texttt{)}, \text{ if } de(n) = [\oplus, n] \\ &\quad \texttt{f(}m, \texttt{0)}, \text{ if } de(n) = [m, n] = de(m) \\ &\quad \texttt{f(}m\texttt{,}\sigma(de(cnda(n)))\texttt{)}, \text{ if } de(n) = [m, n] \neq de(m) \end{aligned}$$

□

Definition 6.3.6 Let $T = (N, r, p, l)$ and $T' = (N', r', p', l')$ be two trees, and let K be a minimal edge cover of their induced graph $IG(T, T')$. Without loss of generality, let $[r, r']$ be the only edge in K incident on either of r or r'. We define the *transformation induced by the cover* K, denoted by $F(K, T, T')$ as follows, where $m, m' \notin \{r, \oplus\}$, $n \notin \{r', \ominus\}$, $E(m) = \{[m, n] \in K\} \cup \{[n, m] \in K\}$, and $cdp(T, K)$ is a short-hand for $cdp(T, T'K) \cap T$. (We use `type font` to represent literal strings, and *italic font* to represent non-literals, and where σ is a function that maps edges to arbitrary, unique, positive integers.)

$$\begin{aligned} F(K, T, T') &= \{\texttt{del(f(}m, \texttt{0))} \mid [m, \ominus] \in K\} \\ &\cup \{\texttt{ins(}h(p(n)), l(n), i) \mid [\oplus, n] \in K\} \\ &\cup \{\texttt{mov(f(}m, \texttt{0)}, h(p(n))) \mid m \in cdp(T, K),\ [m, n] \in K, \\ &\quad |E(m)| = |E(n)| = 1\} \end{aligned}$$

$$
\begin{aligned}
\cup\ & \{\texttt{mov}(\texttt{f}(m,\texttt{0}), h(p(n))) \mid m \in cdp(T,K),\ [m,n] \in K,\ |E(m)| > 1, \\
& de(m) = [m,n],\ de(cnda(p(m))) = [cnda(p(m)), n'] \neq de(n')\} \\
\cup\ & \{\texttt{cpy}(\texttt{f}(m,\texttt{0}), h(p(n)), \sigma([\texttt{m},\texttt{n}])) \mid m \in cdp(T,K),\ [m,n] \in K, \\
& |E(m)| > 1,\ [m,n] \neq de(m)\} \\
\cup\ & \{\texttt{mov}(\texttt{f}(m,\texttt{0}), h(p(n))) \mid m \in cdp(T,K),\ [m,n] \in K,\ |E(n)| > 1, \\
& de(n) = [m,n],\ de(p(n)) \neq [p(m'), p(n)]\} \\
\cup\ & \{\texttt{glu}(\texttt{f}(m,\texttt{0}), \texttt{f}(m',\texttt{0})) \mid m \in cdp(T,K), \\
& [m,n] \in K,\ |E(n)| > 1,\ [m',n] = de(n),\ m \neq m'\} \\
\cup\ & \{\texttt{upd}(h(n), l(n)) \mid [m,n] \in K,\ [m,n] \neq de(m),\ l(m) \neq l(n), \\
& |E(m)| > 1\}
\end{aligned}
$$

□

Example 6.3.1 Let T be the initial tree from Example 6.2.1, and let T' be a tree isomorphic to the final tree $F(T)$ there, as shown in Figure 6.3. (Note that in Example 6.2.1 the final tree is obtained by modifying the initial tree. Here, the two trees are not related in this manner; hence the tree nodes do not share identifiers.) A minimal edge cover of their induced graph is indicated using dashed lines. (Note that this edge cover is in fact the signature of the transformation in Example 6.2.1.) The cover disassembly points, computed using Definition 6.3.3, are marked by asterisks. The distinguished edge incident on each node is marked by a small filled circle on that edge near the corresponding node. Since there are only two nodes with more than one edge incident on them, the choice of a distinguished edge is nontrivial in these two cases only. Intuitively, for node 5, we observe that the parent of node 56 is matched to the parent of node 5; therefore the edge $[5, 56]$ is chosen as distinguished. Node 4 is not matched to any node whose parent matches the parent of node 4; therefore, we select a distinguished edge arbitrarily from those incident on node 4, say $[4, 53]$. (Definition 6.3.4 describes the choice formally.)

Using this information about the cover disassembly points and distinguished edges, we now use Definition 6.3.6 to obtain a transformation. The edge $[4, 54]$ satisfies the

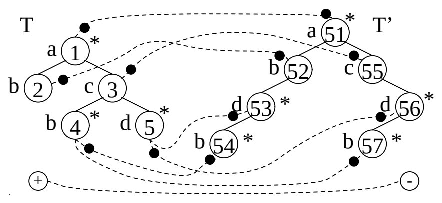


Figure 6.3: The trees in Example 6.3.1

conditions in line 4 of the equation in Definition 6.3.6, resulting in the operation `mov(f(4,0),` $h(p(54))$`)`. Now from Figure 6.3 we observe that $p(54) = 53$. Further, node 53 is matched to node 5, but $[5, 53]$ is not the distinguished edge incident on node 5 (i.e., $de(5) \neq [5, 53]$); therefore, the representative handle of node 53 is `f(5,`$\sigma([5,53])$`)`, where σ is simply a function that generates a unique identifier for each edge. Say $\sigma([5,53]) = 501$, so that $h(p(54)) = h(53) =$ `f(5,501)`, giving `mov(f(4,0), f(5,501))` as the edit operation generated corresponding to edge $[4, 54]$. Next, observe that edge $[5, 53]$ satisfies the conditions in line 6 of Definition 6.3.6, resulting in the operation `cpy(f(5,0), f(2,0), 501)`, since $h(p(53)) = h(52) =$ `f(2,0)`, and $\sigma([5,53]) = 501$. A similar process for the edge $[4, 57]$ results in the operation `cpy(f(4,0), f(5,0))`. Definition 6.3.6 does not generate any more operations, giving {`mov(f(4,0), f(5,501)), cpy(f(5,0), f(2,0), 501), cpy(f(4,0), f(5,0))`} as the recovered transformation.

Observe that the transformation recovered above is essentially identical to that in Example 6.2.1. Apart from edit operation identifiers, the only difference is that instead of moving the node 4 to below the original instance of node 5 (and copying node 4 to below the copy of node 5) as done by that transformation, the above transformation moves node 4 to below the copy of node 5 (copying node 4 to below the original instance of node 5). This difference is a result of the freedom in the choice of a distinguished edge incident on node 4. □

We now state and prove the main results of this chapter as Theorems 5 and 6 below, showing that the transformation F' recovered by Definition 6.3.6 from the signature $S(F, T)$ of a transformation F on tree T is essentially identical to F:

Theorem 5 *Let T and T' be two trees, and let K be a minimal edge cover of their induced graph $IG(T, T')$. Then (1) $F(K, T, T')$ is a valid transformation for T, (2) $F(K, T, T')(T)$ is isomorphic to T', and (3) $S(F(K, T, T'), T)$ is isomorphic to K.* □

Theorem 6 *Let T be a tree, let F be a transformation that is valid for T, and let $F' = F(S(F, T), T, F(T))$. Then F' has the same number of move, copy, and glue operations as F (respectively), and the insert, delete, and update operations in F' are identical to those in F, modulo edit operation identifiers.* □

The following lemma is useful in proving the above theorems:

Lemma 11 *Let $T = (N, r, p, l)$ and $T' = (N', r', p', l')$ be two trees, and let K be a minimal edge cover of their induced graph $IG(T, T')$. If $m, n \notin cdp(T, K)$, $m \neq \ominus$, $m \neq r$, $n \neq \oplus$, and $n \neq r'$, then $[p(m), p'(n)] \in K$. Consequently, we have $[cnda(m), cnda(n)] \in K$ for all nodes $m \in N$, $n \in N'$.* □

Proof Follows from Definition 6.3.3. □

Proof of Theorem 5

Part (1): Let us first show that $F = F(K, T, T')$ is a valid transformation for T. The conditions 1(a-e) and 2 in Definition 6.2.2 are easy to verify. Let us consider condition 3 for some node n such that $del(n) \in F$, and any child c of n. If $[c, \ominus] \in K$, $del(f(c, 0)) \in F$. Otherwise $[c, y] \in K$ for some $y \in T'$. Since the only edge in K incident on $n = p(c)$ is $[n, \ominus]$ (due to minimality of K and Definition 6.3.2), it follows from Definition 6.3.3 that $m \in cdp(T, K)$. Definition 6.3.6 shows that every node $m \in cdp(T, K)$ such that $[m, \ominus] \notin K$ is acted on by a *mov* or *glu* operation in F. Thus condition 3 is satisfied. Finally, let us verify condition 1(f) of Definition 6.2.2. If $glu(f(m_1, 0), f(m_2, 0), i) \in F$ then $[m_1, n], [m_2, n] \in K$ due to Definition 6.3.6. Let x be any node in $dc(m_1)$, implying that m_1 is an ancestor of x. Consider first the case when $p(x) = m_1$. Since $x \notin dp(T, F)$, it follows from Lemma 10 and Definition 6.3.3 that $\exists! [x, y] \in K$ such that $p(y) = n$. (We use $\exists!$ to denote "there exists a unique.") Furthermore, $x \notin dp(T, K)$ implies $y \notin dp(T, F) = cdp(T, K)$, so that $\exists! [x', y] \in K$ such that $p(x') = m_2$. Thus for any child x of m_1, we determine uniquely a corresponding child x' of m_2, and we define $g_i(x) = x'$. By using induction on the depth of a node x in the subtree $dc(m_1)$, we extend the definition of g_i to all nodes in $dc(m_1)$. We thus have a one-to-one function $g_i : dc(m_1) \rightarrow dc(m_2)$ that preserves the parent function; by symmetry, it follows that g_i is also an onto function. Finally, it is easy to verify that g_i "preserves labels" in the sense of Definition 6.2.2. Therefore F is a valid transformation for T.

Part (2): Let us now show that $F(T) = (N'', r'', p'', l'')$ is isomorphic to $T' = (N', r', p', l')$. Define a function $h' : N' \rightarrow N''$ intuitively reflecting the representative handle function h in Definition 6.3.5 as follows:

$$\begin{aligned} h'(n) \;\; &= \;\; f'(0,i), \text{ if } h(n) = f(0,i) \\ & \;\;\;\;\;\; f'(m,0), \text{ if } h(n) = f(m,0) \\ & \;\;\;\;\;\; f'(m,i), \text{ if } h(n) = f(m,i) \end{aligned}$$

We claim that h' is an isomorphism from $F(T)$ to T'. Since there is exactly one distinguished edge $de(n)$ incident on any node $n \in F(T)$, it follows that h, and hence h', is a **one-to-one** function.

Now let us show that h' is an **onto** function. Consider any $m' \in F(T)$, and the possibilities according to Definition 6.2.3. If $m' = f'(0,i)$, we know that $ins(\ldots, i) \in F(T)$, implying $de(n) = [\oplus, n]$ and $\sigma(de(n)) = i$, so that $h(n) = f(0,i)$ and $h'(n) = m'$. If $m' = f'(m,0)$, we know $m \in T$ and $del(f(m,0)), glu(cnda(m), \ldots) \notin F$. Thus $[m, \ominus] \notin S(F,T)$ and $\exists n = cnda(n) \in T' : de(cnda(m)) = [cnda(m), n] = de(n)$ (since $de(cnda(m)) \neq de(cnda(n))$ implies $glu(f(cnda(m),0), \ldots) \in F$), implying $h(n) = f(m,0)$ and $h'(n) = f'(m,0) = m'$. Finally, if $m' = f'(m,i)$, $i > 0$ then $cpy(f(cnda(m),0), ...) \in F$, so that $\exists n' \in T' : de(n') = [cnda(m), n] \neq de(cnda(m))$. Now using the definition of $cdp(T,K)$ and the fact that $m \notin cdp(T,K)$, it is easy to observe that $\exists n \in cdc(n') : de(n) = [m,n] \neq de(m)$, implying $h(n) = f(m,i)$ and $h'(n) = f'(m,i)$.

We shall now show that h' **preserves the parent function**; that is, we shall show that $h'(p'(n)) = p''(h'(n))$ for all $n \in N'$, $n \neq r'$. If $h'(n) = f'(0,i)$ then $ins(h(p(n)), l(n), i) \in F$, implying $p''(f'(0,i)) = h'(p'(n))$ as needed. If n (and therefore $h(n)$) is not a disassembly point, clearly $p''(h'(n)) = h'(p'(n))$ by Definition 6.2.3.

If n (and therefore $h'(n)$) is a disassembly point, we have $n = cnda(n)$, $m = cnda(m)$, and the following two cases:
Case 1: $h'(n) = f'(m, 0)$. In this case, $h(n) = f(m, 0)$, implying $de(n) = [m, n]$ and $de(m) = de(n)$ (since m and n are disassembly points) by the definition of the handle function h. Using the definition of $F(K, T, T')$, it is easy to observe that for all the three of the possibilities (1) $|E(n)| = |E(m)| = 1$, (2) $|E(n)| = 1,\ |E(m)| > 1$, and (3) $|E(n)| > 1,\ |E(m)| = 1$, $mov(f(m, 0), h(p(n))) \in F$, implying $p''(f'(m, 0)) = h'(p(n))$ as needed.
Case 2: $h'(n) = f'(m, i),\ i > 0$. In this case, $cpy(f(m, 0), h(p(n))) \in F$, implying $p''(f'(m, i)) = h'(p(n))$ as needed.

From the definition of $h'(n)$, $h(n)$, and $F(K, T, T')$, it is easy to see that h' **preserves the label function** l. Thus, h' is the required isomorphism between T' and $F(T)$.

Part (3): Now let us show that $S = S(F, T)$ is isomorphic to K; more precisely, we show that $[m, n] \in K$ if and only if $[m, h'(n)] \in S(F, T)$. (We extend h' by defining $h'(\ominus) = \ominus$ for notational convenience.) It is easy to observe that $[\oplus, \ominus] \in K$ if and only if $[\oplus, \ominus]$ in S; therefore we will exclude this special edge from our discussion below.

Consider any $[m, n] \in K$. We will show that $[m, h'(n)] \in S$. If $m = \oplus$, $ins(\ldots, i) \in F$, so that $[\oplus, f'(0, i)] = [m, h'(n)] \in S$ as required. Otherwise, we have two cases:
Case 1: $h'(n) = f'(m, 0)$. If $de(n) = [m', n],\ m' \neq m$ then (by lemma Lemma 11) $[cnda(m), cnda(n)]$ and $[cnda(m'), cnda(n)]$ belong to K, in turn implying $glu(f(cnda(m), 0), f(cnda(m'), 0))$ so that $[m, f'(m, 0)] \in S$. Otherwise $de(n) = [m, n]$, implying $glu(f(m, 0), \ldots) \notin F$. Now if $\exists m' \in N : glu(f(m', 0), f(m, 0)) \in F$, we can argue $[m, f'(m, 0)]$ as before; else the absence of glu and del operating on m gives $[m, f'(m, 0)]$ as required.
Case 2: $h'(n) = f'(n, i),\ i > 0$. From the definition of the representative handle function h and the copy operation-generating part of the definition of $F(K, T, T')$ (Definition 6.3.5), we see that $\exists cpy(f(cnda(m), 0), m', i) \in F$. Therefore, $[m, f'(m, i)] \in S$ as required.

Thus $[m, n] \in K \Rightarrow [m, h'(n)] \in S$. Now since $h' : N' \to N''$ is an isomorphism, all edges in S are of the form $[m, h'(n)]$, where $m \in N \cup \{\oplus\}$ and $n \in N' \cup \{\ominus\}$. Consider any such edge. If $m = \oplus$ then $h'(n) = f'(0, i)$, implying $[\oplus, n] \in K$ by Definition 6.3.5. If $h'(n) = f'(m, 0)$ then $h'(n) = m$ gives $[m, n] \in K$. Otherwise, $h'(n) = f'(m, i)$, $i > 0$, implying $[m, n] \in K$ again. Thus $[m, h'(n)] \in S \Rightarrow [m, n] \in K$, which together with $[m, n] \in K \Rightarrow [m, h'(n)] \in S$, shows that S and K are isomorphic. □

Proof of Theorem 6

Let $T = (N, r, p, l)$, $T' = F(T) = (N', r', p', l')$. Consider first any **insert** operation $ins(h(p'(n)), l(n), i)$ in F'. From the definition of $F(K, T, T')$, we know that $[\oplus, n] \in S(F, T)$ such that $\sigma([\oplus, n]) = i$, which in turn implies $ins(h, l, i) \in F$ and $n = f'(0, i)$ due to Definition 6.3.1. Since $l(f'(0, i)) = l$, it follows that $l = l(n)$. If $h = f(n_2, 0)$, $p'(n) = f'(n_2, 0)$; else $h = f(n_2, i)$ and $p'(n) = f'(n_2, i)$; in either case, $h = h(p'(n))$. Thus $ins(h(p'(n)), l(n), i) \in F$.

Consider any **delete** operation $del(f(m, 0)) \in F'$. From the definition of $F(K, T, T')$, we obtain $[m, \ominus] \in S(F, T)$, which in turn implies $del(f(m, 0)) \in F$ due to Definition 6.3.1. The above arguments for insert and delete operations can also be repeated in the reverse direction.

Consider a node $m \in T$ such that $|E(m)| = k > 1$. Let the set of edge-cover edges incident on m be $E(m) = \{[m, n_i]\}_{i=1}^{k}$. Now since the edge cover is actually $S(F, T)$, Definition 6.3.1 implies that there are exactly $k - 1$ copy operations of the form $cpy(f(m, 0), h)$ in F. It is easy to observe that the definition of $F(K, T, T')$ generates exactly $k - 1$ copy operations for such a node m. Since the above argument can be repeated for each node $m \in T$, we conclude that the number of **copy** operations in F is equal to that number in F'.

The argument for **glue** operations is analogous to the above argument for copy operations: Consider a node $n \in T'$ such that $|E(n)| = k > 1$. Let the set of edge-cover edges incident on n be $E(n) = \{[m_i, n]\}_{i=1}^{k}$. Now since the edge cover is actually $S(F, T)$, Definition 6.3.1 implies that there are exactly $k - 1$ glue operations of the form $glu(f(m_j, 0), f(m', 0))$ in F, where $m' \in \{m_i\}$ and $m_j \in \{m_i\}$ for $j = 1 \ldots k - 1$. It is easy to observe that the definition of $F(K, T, T')$ generates exactly $k - 1$ glue operations corresponding to such a node n. Since the above argument can be repeated for each node $n \in T'$, we conclude that the number of glue operations in F is equal to that number in F'.

Now consider **move** operations. Consider first any move operation in the third subset of the definition of $F(K, T, T')$. We know that $m \in cdp(T, K)$, so that $m \in dp(T, F)$ by Lemma 10. From Definition 6.2.1, we see that a non-root node is in $dp(T, F)$ only if it is acted on by some edit operations other than update. Now, m cannot be deleted, because that would imply $[m, \ominus] \in S(F, T)$. Furthermore, the fact that $|E(m)| = |E(n)| = 1$ indicates that m cannot be subject to a copy or a glue operation. Consequently, it must be the case that m is moved by F.

Now consider any move operation $mov(f(m, 0), h(p(n)))$ in the fourth subset of the definition of $F(K, T, T')$. We know that $|E(m)| > 1$; let $E(m) = \{[m, n_i]\}_{i=1}^{k}$ where $k > 1$. As we have seen above, there are $k - 1$ copy operations of the form $cpy(f(m, 0), \eta)$ in F. Let $[m, n^*]$ be the unique edge in $E(m)$ that does not correspond to a copy operation. Suppose $mov(f(m, 0), \eta) \notin F$. Then $p'(f'(m, 0)) = f'(p(m), 0)$ by Definition 6.2.3. Now if either $del(f(p(m), 0)) \in F$ or $glu(f(p(m), 0), h') \in F$, the validity of F implies $mov(f(m, 0), h) \in F$ (due to Definition 6.2.2), contradicting our assumption. Therefore, it must be the case that $del(f(p(m), 0)) \notin F$ and $glu(f(p(m), 0), h') \notin F$, in turn implying $f'(p(m), 0) \in N'$ due to Definition 6.2.3. Now, using Definition 6.3.1, the above facts yield $[p(m), f'(p(m), 0)] \in S(F, T) = K$, in turn implying $E'(m) \neq \emptyset$ in Definition 6.3.6, which gives $de(m) \in E'(m)$, contradicting the condition in the fourth subset of Definition 6.3.6. Therefore, we conclude that F contains a move operation $mov(f(m, 0), h)$. The argument for move operations in the sixth subset of the definition of $F(K, T, T')$ is analogous to the above.

Finally, let us consider **update** operations. Consider first an edge $[m, n] \in S(F, T)$ such that $|E(m)| = |E(n)| = 1$, implying $n = f'(m, 0)$. Since $l(n) \neq l(m)$, clearly $upd(f(m, 0), l(n)) \in F$ due to Definition 6.2.3. Now consider an edge $[m, n] \in S(F, T)$ such that $|E(m)| = 1$ and $|E(n)| = k > 1$, implying $k-1$ glue operations corresponding to $k - 1$ of the k edges in $E(n)$. Now if $[m, n]$ is the edge not corresponding to a glue operation, we can argue as above that $upd(f(m, 0), l(n)) \in F$. On the other hand, if $[m, n]$ is an edge corresponding to a glue operation $glu(f(m, 0), f(m', 0))$, the condition 1(f) in Definition 6.2.2 requires $upd(f(m, 0), l(n)) \in F$. We have thus shown that all the update operations in $F'(K, T, T')$ for nodes m such that $|E(m)| = 1$ are also present in F. Now consider an edge $[m, n] \in S(F, T)$ such that $|E(m)| = k > 1$ and $|E(n)| = 1$, implying $k - 1$ copy operations corresponding to $k - 1$ of the k edges in $E(n)$. Now if $[m, n]$ is the edge not corresponding to a copy operation, we can argue as above that $upd(f(m, 0), l(n)) \in F$. On the other hand, if $[m, n]$ is an edge corresponding to a copy operation $cpy(f(m, 0), h, i)$, $n = f'(m, i)$, implying $upd(f(m, i), l(n)) \in F$ since $l(m) \neq l(n)$ (by Definition 6.2.3). It is easy to show that the argument for update operations also holds in the reverse direction; that is, an update operation in F implies a corresponding one in F'. $\square$

6.4 Computing Signatures

In this section, we briefly describe the application of our ideas presented in earlier sections. In particular, we outline the benefits of our transformation model, and describe how signatures can be used to efficiently compute a minimum-cost transformation between two trees. When managing tree-structured data (e.g., structured query results, programs, documents, Web sites, circuit designs, and file systems), one often needs to find differences between related data (e.g., results of running a query at different times, two similar circuits, or different versions of a program or document). Such tree differences can be compactly and effectively captured by the novel transformation model we have presented here. Our model includes expressive subtree operations, such as move and copy, which make the detected differences more meaningful to a user. This model also admits representative signatures, which are compact representations of the essential information in transformations. These signatures make it possible to search for a minimum-cost transformation by searching instead for a minimum-cost signature, knowing that each signature can be mapped back to a transformation. As discussed in Section 6.1 and below, working with transformation signatures greatly simplifies algorithms for computing minimum-cost transformations. Our model, results, and strategy for computing a minimum-cost transformation are independent of the details of the cost model used. Furthermore, although in this chapter we have focused on unordered trees, the results adapt easily to ordered trees, making our scheme widely applicable.

A general approach to computing a minimum-cost signature, without using application- or domain-specific features, is to use search-based techniques and heuristics. Recall from Section 6.3 that the signature of any transformation between the input trees T_1 and T_2 is a minimal edge cover of their induced graph (which is a bipartite graph that has an edge between every node in T_1 and every node in T_2). Thus, the search space that we need to explore is the space of all possible minimal edge covers of this bipartite graph. Note that this is a much simpler search space than the search space of all possible transformations between T_1 and T_2. This simplification is a result of the existence of representative signatures in our transformation model.

Further, we can use *pruning rules*, such as those introduced in Chapter 5, to eliminate edges from the induced graph. Recall that these rules (conservatively) detect cases when two nodes can never be partners in any minimum-cost transformation (using upper and lower bounds on the contribution of an induced graph edge to the cost of a signature). (In addition, we may optionally decide to use aggressive pruning rules that prune edges if it is "very unlikely" that the corresponding nodes could be partners.) Eliminating edges from the induced graph greatly reduces the size of the search space. Next, we use estimates of the cost contribution of induced graph edges to compute a minimum-(estimated)cost edge cover of the pruned induced graph. Finally, we search for a better signature in the neighborhood of this initial edge cover, using techniques similar to those in [WZC95, SWZS94]. In Chapter 9, we present experimental results that explore some of these options.

We can often further reduce the size of the search space of signatures by using features of the application domain. For example, consider an application comparing structured documents. Such documents are often represented using layered, ordered trees, with layers corresponding to structural elements (such as words, sentences, paragraphs, and sections). That is, each tree node has an immutable type, and the tree is layered by a partial order on these types. (For example, sentences are below paragraphs and sections.) Therefore we do not need to consider any signature that matches nodes of different types to each other. This fact leads to very effective pruning of the induced graph, and a corresponding reduction in the size of the search space of its minimal covers.

Due to the simplicity of the relation between signatures and transformation in our model, we are able to derive tighter bounds on the edge costs described above than those possible in the linear edit script model. (With linear edit scripts, cost bounds need to take into account possible "piggy-backing" of edit operations.) These tighter bounds lead to more effective pruning of the induced graph, and thus give us better performance. This simplicity also allows us to easily derive better estimates for edge costs in the induced graph, thus improving the quality of the initial solution, and the effectiveness of the subsequent search process.

Finally, we can often use domain characteristics in conjunction with the properties of representative signatures to permit the exact computation of the contribution of an edge in the induced graph to the total cost of the signature (whereas in general we use estimates). Consequently, the initial solution that was earlier the estimated minimum-cost signature is now the actual optimal solution, so that the subsequent search phase is unnecessary. One such scheme (similar in spirit to the restrictions on matchings used in [Yan91, ZS89, ZWS95]) results in a simple bottom-up dynamic programming algorithm that produces optimal solutions if moves, copies, and glues are restricted to be "local." (A restriction of local copies, for instance, disallows a paragraph from being copied outside its section.) Even if these restrictions do not strictly hold in a given application domain, we may intuitively expect such algorithms produce solutions that are close to optimal.

6.5 Summary

In this chapter, we described the difficulties encountered when we use the traditional linear edit script model with expressive subtree operations such as move, copy, and uncopy. To address these difficulties, we presented a novel model for describing tree transformations. Unlike edit scripts, which constitute a procedural specification of the differences between two trees, transformations in our model provide a simple declarative specification of tree differences. Intuitively, our transformations operate on a tree by first dividing the tree into components called chunks, then operating on these chunks independently of one another, and finally putting the chunks together to form the final tree.

The essential features of the tree transformations defined in this chapter are compactly represented by their signatures. We defined representative signatures of transformations and presented the simple algorithms used to map transformations and signatures to each other. Due to the declarative nature of our transformations and the lack of restrictions such as those required for structured edit scripts in Chapter 5, these algorithms are extremely simple. Our algorithm to map signatures to transformations, given by Definition 6.3.6, is substantially simpler than the analogous algorithm for structured edit scripts presented in Section 5.4.3 of Chapter 5.

We also described how some of the techniques used for the linear edit script model, including those described in earlier chapters, can be adapted to this transformation model. In particular, we have implemented a program that combines the pruning techniques from Chapter 5 with the transformations of this chapter. In Chapter 9, we describe experimental results based on this implementation.

Recall, from Chapter 3, that detecting changes by comparing data snapshots is an important subproblem of the problem of managing change in heterogeneous, autonomous databases. In this chapter and Chapters 4 and 5, we studied several change detection techniques. In the next chapter, we explore how such changes, once detected, can be represented, stored, and queried in a systematic manner, and in Chapter 8 we describe how we combine these ideas in the implementation of the C^3 system.

Chapter 7

Representing and Querying Changes

In Chapters 4, 5, and 6, we described techniques for detecting changes in heterogeneous, autonomous databases. In this chapter, we address the issue of how these changes are stored, queried, and managed. Recall from Chapter 3 that the data we are interested in is semistructured in nature. The lack of a fixed schema inherent in such data makes it very difficult to use traditional database techniques for representing and querying historical data. We therefore present a simple and general model, DOEM (pronounced "doom"), for representing changes in semistructured data. We also present a language, *Chorel*, for querying over data and changes represented in DOEM. We describe our implementation of DOEM and Chorel. We also introduce a facility that allows users to subscribe to changes in semistructured data, and we describe its design and implementation based on DOEM and Chorel.

7.1 Introduction

Recall from Chapter 3 that *semistructured data* is data that has some structure, but it may be irregular and incomplete and does not necessarily conform to a fixed schema. Recently, there has been increased interest in data models and query languages for semistructured data [Abi97, BDHS96, CACS94, CGMH+94, QWG+96].

We also see increased interest in *change management* in relational and object data [GHJ96, DHR96], and in the related problem of *temporal databases* [SA86, Soo91]. However, we are not aware of any work that addresses the problem of representing and querying changes in semistructured data. As will be seen, this problem is more challenging than the corresponding problem for structured data due to the irregularity, incompleteness, and lack of schema that often characterize semistructured data. Nevertheless, our approach, based on graph annotations, is also applicable to structured graph-based data.

7.1.1 Motivating Examples

The *Palo Alto Weekly*, a local newspaper, maintains a Web site providing information about restaurants in the Bay Area [PAW98]. Most of the data in the restaurant guide is relatively static. But as often happens in database applications, we are particularly interested in the dynamic part of the data. For example, we are interested in finding out which restaurants were recently added, which restaurants were seen as improving, degrading, and so on. These changes can be captured using the differencing techniques described in Chapters 4, 5, and 6. Figure 3.3 in Chapter 3 depicts some sample output produced by our differencing program on inputs from the Palo Alto Weekly. (Our program, *Tdiff*, is described in detail in Chapter 8.)

For reasonably small documents, browsing the marked-up HTML files produced by *htmldiff* to view the changes of interest is a feasible option. However, as documents get larger and changes become more prevalent and varied, one soon feels the need to use queries to directly find changes of interest instead of simply browsing. (For example, the restaurant guide page is currently more than 20,000 lines long, making browsing very inconvenient.) An example of a simple change query over the restaurant data is “find all new restaurant entries.” Another example is “find all restaurants whose average entree price changed.” Just as browsing databases is often an ineffective way to retrieve information, the same holds for browsing data representing changes. Thus, for this example, what we need is a query language that allows queries over changes to (semistructured) HTML pages.

As another motivating example, consider a typical library system that contains book circulation information. Suppose we wish to be notified whenever any "popular" book becomes available where, say, we define a book as popular if it has been checked out two or more times in the past month. We could partially achieve this goal by setting a trigger on the circulation database that notifies us whenever a book is returned. However, there are two problems with this approach. First, many library information systems are legacy mainframe applications on which triggers are not available. Furthermore, even in cases where the library information system is implemented using a database system that supports triggers, a user often lacks the access rights required to set triggers on the database. Second, there is often no way to access historical circulation information, so that we cannot check whether the book being returned was checked out two or more times recently. In this application too, the data may be semistructured, especially if the library system merges information from multiple sources [PAGM96]. Thus, we again need a method to compute, represent, and query changes in the context of semistructured data.

7.1.2 Overview

Since our goal is to represent changes in semistructured data, we use as a starting point the *Object Exchange Model* (OEM), designed at Stanford as part of the Tsimmis project [CGMH+94]. OEM is a simple graph-based data model, with objects as nodes and object-subobject relationships represented by labeled arcs. Due to its simplicity and flexibility, OEM can encode numerous kinds of data, including relational data, electronic documents in formats such as SGML and HTML, other data exchange formats (e.g., ASN.1), and programs (e.g., C++). Note that OEM may be thought of as an extension of the tree-based models used in Chapters 4, 5, and 6 to directed graphs. The basic change operations in such a graph-based model are node insertion, update of node values, and addition and removal of labeled arcs. (Node deletion is implicit by unreachability [AQM+96].) Our change representation model, DOEM (for *Delta*-OEM), uses *annotations* on the nodes and arcs of an OEM graph to represent changes. Intuitively, the set of annotations on a node or arc represents the history of that node or arc.

For querying over changes we use a language based on the *Lorel* language for querying semistructured data [AQM+96]. In our language, called *Chorel* (for *Change Lorel*), we extend the concept of Lorel *path expressions* to allow us to refer to the annotations in a DOEM database. The result is an intuitive and convenient language for expressing change queries in semistructured data. Although the work in this chapter is founded on the OEM data model and the Lorel language, the principal concepts are applicable to any graph-based data model (semistructured or otherwise), e.g., [BDHS96, Cat96].

Our implementation of DOEM and Chorel uses a method that encodes DOEM databases as OEM databases and translates Chorel queries into equivalent Lorel queries over the OEM encoding. This encoding scheme has the benefit that we did not need to build from scratch yet another database system; instead, we capitalized on an existing database system for semistructured data. Finally, as an important first application of DOEM and Chorel, we describe our design and implementation of a *query subscription service* that permits users to subscribe to changes in semistructured data.

7.1.3 Contributions

The main contributions of this chapter are as follows:

1. We present a simple and general change representation model for semistructured data. An important feature of our model is that it represents changes to a database directly as graph annotations, instead of indirectly as the difference between old and new database states.

2. We describe the syntax, semantics, and implementation of a query language over changes to semistructured data. Again, an important advantage of our query language is that it allows the user to access changes directly.

3. We describe how our system implements this change query language on top of an existing semistructured database system by encoding the change data and by translating change queries to ordinary queries.

4. We show how "virtual annotations" can be used to access implicit information in our data model. In particular, we describe how our query language (and its translation-based implementation) is extended to facilitate snapshot-based access to data.

5. We describe the design and implementation of a *query subscription service* that permits users to subscribe to changes in heterogeneous database environments. A unique feature of our service is that it enables the user to specify very precisely (using our query language) the changes of interest.

The rest of this chapter is organized as follows. Section 7.2 reviews the Object Exchange Model (OEM), and introduces OEM change operations and histories. In Section 7.3, we present our OEM-based change representation model for semistructured data, DOEM. Section 7.4 describes our change query language, Chorel. In Section 7.5, we present the encoding scheme that we use to implement DOEM and Chorel by translation, and we briefly describe our system implementation. In Section 7.6, we introduce some extensions to our language that make snapshot-based access in our data model more convenient. We also describe how our translation-based implementation of Chorel is extended for this purpose. Section 7.7 describes the query subscription system we have implemented based on the material in Sections 7.3–7.5. We conclude in Section 7.8.

7.2 The Object Exchange Model

The *Object Exchange Model (OEM)* is a simple, flexible model for representing heterogeneous, semistructured data. In this section, we begin by briefly describing OEM. Next, we define the basic change operations used to modify an OEM database. Finally, we introduce the concept of an OEM *history* that describes a collection of basic change operations. Histories form the basis of our change representation model described in Section 7.3.

Intuitively, one can think of an OEM database as a directed graph in which nodes correspond to objects and arcs correspond to relationships. Each arc has a label

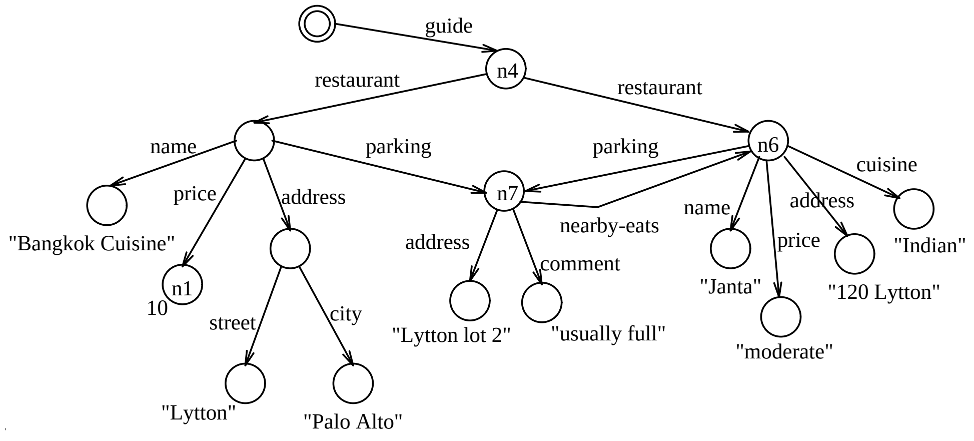


Figure 7.1: The OEM database in Example 7.2.1.

that describes the nature of the relationship. (Note that the graph can have cycles, and that an object may be a subobject of multiple objects via different relationships. Example 7.2.1 below illustrates these points.) Nodes without outgoing arcs are called *atomic objects*; the rest of the nodes are called *complex objects*. Atomic objects have a *value* of type integer, real, string, etc. An arc (p, l, c) in the graph signifies that the object with identifier c is an l-labeled subobject (child) of the complex object with identifier p. Each OEM database has a distinguished node called the *root* of the database. The root is the implicit starting point of path expressions in the Lorel query language (described in Section 7.4.1). Formally, we define an OEM database as follows:

Definition 7.2.1 An OEM *database* is a 4-tuple $O = (N, A, v, r)$, where N is a set of object identifiers; A is a set of labeled, directed arcs (p, l, c) where $p, c \in N$ and l is a string; v is a function that maps each node $n \in N$ to a value that is an integer, string, etc., or the reserved value $\mathcal{C}$ (for complex); and r is a distinguished node in N called the *root* of the database. A node is a *complex object* if its value is $\mathcal{C}$ and otherwise it is an *atomic object*. Only complex objects have outgoing arcs. We also require that every node be reachable from the root using a directed path. $\square$

Example 7.2.1 We will use as our running example an OEM database describing the restaurant guide section of the *Palo Alto Weekly*, introduced in Section 7.1. Figure 7.1 shows a small portion of the data. Note that although the restaurant entries are quite similar to each other in structure, there are important differences that require the use of a semistructured data model such as OEM. In particular, we see that the price rating for a restaurant may be either an integer (10) or a string ("moderate"). The address may be either a simple string ("120 Lytton") or a complex object with subobjects listing the street, city, etc. Note also that although the data has a natural hierarchical structure, nodes may have multiple incoming arcs (e.g., node n_7), and there are cycles (e.g., the cycle formed by the arcs "parking" and "nearby-eats"). In the sequel, we refer to this database as *Guide*. □

7.2.1 Changes in OEM

We now describe how an OEM database is modified. Let $O = (N, A, v, r)$ be an OEM database. The four *basic change operations* are the following:

Create Node: The operation *creNode*(n, v) creates a new object. The identifier n must be new, i.e., n must not occur in O. The initial value v must be an atomic value (integer, real, string, etc.) or the special symbol $\mathcal{C}$ (for complex).

Update Node: The operation *updNode*(n, v) changes the value of object n, where v is an atomic value or the special symbol $\mathcal{C}$. Object n must be either an atomic object or a complex object without subobjects. (The model requires us to remove all subobjects of a complex object n before transforming it into an atomic object.) The value v becomes the new value of n.

Add Arc: The operation *addArc*(p, l, c) adds an arc labeled l from object p (the parent) to object c (the child). Objects p and c must exist in O, p must be complex, and the arc (p, l, c) must not already exist in O.

Remove Arc: The operation *remArc*(p, l, c) removes an arc. Objects p and c must exist in O, and O must contain an arc (p, l, c), which is removed.

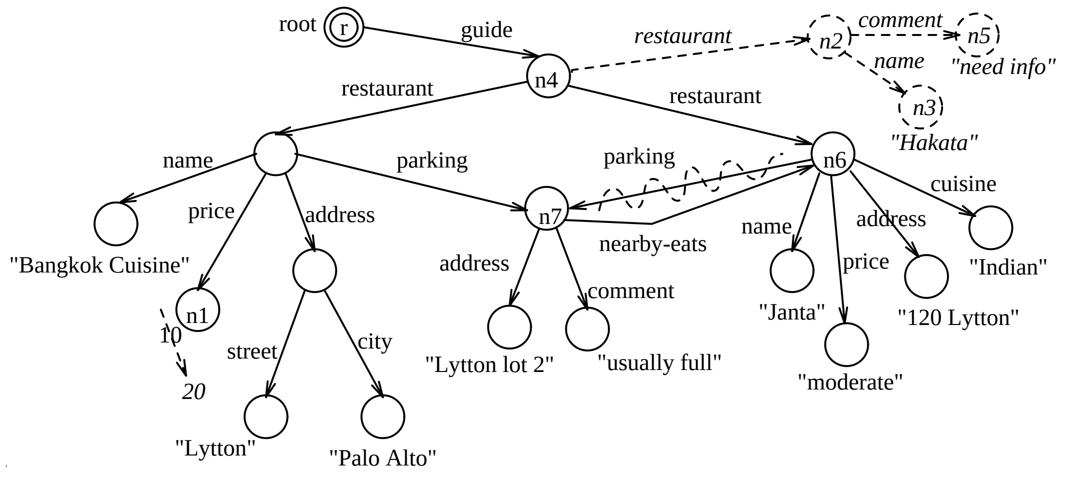


Figure 7.2: The OEM database in Example 7.2.2

If u is a basic change operation that can be applied to O, we say u is *valid* for O, and we use $u(O)$ to denote the result of applying u to O. Note that there is no explicit object deletion operation. In OEM, persistence is by reachability from the distinguished root node [AQM$^+$96]. Thus, to delete an object it suffices to remove all arcs leading to it. A subtlety is that sometimes we need to allow objects to be "temporarily" unreachable. In particular, when we create a new object, it remains unreachable until we create an arc that links it to the rest of the database. Thus, when we consider sequences of changes in Section 7.2.2, we want to permit the result of atomic changes to (temporarily) contain unreachable objects. The issue is discussed further in Section 7.2.2 below. Note that users will typically request "higher-level" changes based on the Lorel update language [AQM$^+$96]; the basic change operations defined here reflect the actual changes at the database level.

Example 7.2.2 Let us consider some modifications to the OEM database in Example 7.2.1. We will use these modifications as a running example in the rest of the chapter. First, on January 1st, 1997, the price rating for "Bangkok Cuisine" is changed from 10 to 20. This modification corresponds to an *updNode* operation. On the same day, a new restaurant with name "Hakata" is added (with no other data). This modification corresponds to two *creNode* operations for the restaurant node and its subobject, and two *addArc* operations to add arcs labeled "restaurant" and "name." Next, on January 5th, a subobject with value "need info" is added to the "Hakata" restaurant object via an arc labeled "comment." This modification corresponds to one *creNode* operation and one *addArc* operation. Finally, on January 8th the parking at "Lytton lot 2" is no longer considered suitable for the restaurant "Janta," and the corresponding arc is removed; this modification corresponds to a *remArc* operation. The resulting modified OEM representation of the Guide data is shown in Figure 7.2, with new data highlighted in bold, and the deleted arc represented using a dashed arrow. □

7.2.2 OEM Histories

We are typically interested in collections of basic change operations, which describe successive modifications to the database. In Chapters 4, 5, and 6, we used sequences of edit operations, called edit scripts, to model collections of edit operations. Below, we define histories, which are generalizations of edit scripts to OEM. Histories differ from the edit scripts of earlier chapters in two major ways: (1) Histories are composed of operations that edit directed graphs (OEM) instead of trees. (2) In order to allow the temporary presence of objects that are unreachable from the root of an OEM database, we divide the operations in a history into sections which may be informally thought of as transactions. (Recall that any object not reachable from the root of an OEM database is implicitly deleted.)

We say that a *sequence* $L = u_1, u_2, \ldots, u_n$ of basic change operations is *valid* for an OEM database O if u_i is valid for O_{i-1} for all $i = 1 \ldots n$, where $O_0 = O$, and $O_i = u_i(O_{i-1})$, for $i = 1 \ldots n$. We use $L(O)$ to denote the OEM database obtained by applying the entire sequence L to O. Also, we say that a *set* $U = \{u_1, u_2, \ldots, u_n\}$ of basic change operations is *valid* for an OEM database O if (1) for some ordering L of the changes in U, L is a valid sequence of changes, (2) for any two such valid sequences L and L', $L(O) = L'(O)$, and (3) U does not contain both $addArc(p, l, c)$ and $remArc(p, l, c)$ for any p, l, and c. We use $U(O)$ to denote the OEM database obtained by applying the operations in the set U (in any valid order) to O.

We are now ready to define an OEM history. Assume we are given some time domain **time** that is discrete and totally ordered; elements of **time** are called *timestamps*. Intuitively, consider an OEM database to which, at some time t_1, a set U_1 of basic change operations is applied, then at a later time t_2, another set U_2 is applied, and so on. A history represents such a sequence of sets of modifications.

Definition 7.2.2 An OEM *history* is a sequence $H = (t_1, U_1), \ldots, (t_n, U_n)$, where U_i is a set of basic change operations and t_i is a timestamp, for $i = 1 \ldots n$, and $t_i < t_{i+1}$ for $i = 1 \ldots n-1$. We say H is *valid* for an OEM database O if, for all $i = 1 \ldots n$, U_i is valid for O_{i-1}, where $O_0 = O$, and $O_i = U_i(O_{i-1})$ for $i = 1 \ldots n$. □

We now return to the requirement that all objects in an OEM database must be reachable from the root. An OEM history can be viewed as a sequence $L_1, ..., L_n$ of sequences of atomic changes. Within one sequence L_i of changes, we relax the requirement that all objects are reachable from the root so that we can, e.g., create a node and then create arcs leading to it, as discussed earlier. However, immediately after each sequence L_i has been applied, nodes that are unreachable are considered as deleted, and the remainder of the history should not operate on these objects. To simplify presentation, we also assume that object identifiers of deleted nodes are not reused.

Example 7.2.3 The history for the modifications described in Example 7.2.2 consists of three sets of basic change operations. It is given by $H = ((t_1, U_1), (t_2, U_2), (t_3, U_3))$, where $t_1 = \mathit{1Jan97}$, $t_2 = \mathit{5Jan97}$, $t_3 = \mathit{8Jan97}$, and:

$$\begin{aligned}
U_1 &= \{\mathit{updNode}(n_1, 20), \mathit{creNode}(n_2, \mathcal{C}), \mathit{creNode}(n_3, \text{``}\mathit{Hakata}\text{''}), \\
&\qquad \mathit{addArc}(n_4, \text{``}\mathit{restaurant}\text{''}, n_2), \mathit{addArc}(n_2, \text{``}\mathit{name}\text{''}, n_3)\} \\
U_2 &= \{\mathit{creNode}(n_5, \text{``}\mathit{need\ info}\text{''})\mathit{addArc}(n_2, \text{``}\mathit{comment}\text{''}, n_5)\} \\
U_3 &= \{\mathit{remArc}(n_6, \text{``}\mathit{parking}\text{''}, n_7)\}.
\end{aligned}$$

This above history is valid for the OEM database of Figure 7.1. □

7.3 Representation of Changes

In Chapter 4, we described a simple structure, called a delta tree, for representing changes between two versions of tree-structured data. Given two trees and the differences (edit script) between them, we produce a delta tree corresponding to each tree by annotating each tree node with the edit operations acting on that node. In this section, we generalize delta trees for representing changes in directed graphs (not only trees) and for representing changes across multiple versions (not only two) of the database As in delta trees, we represent changes to an OEM database by attaching *annotations* to the OEM graph, thereby turning it into a DOEM (*Delta* OEM) graph. We first introduce the annotations we use and define a DOEM database as an OEM graph containing these annotations. Next, we describe how an OEM history (defined in Section 7.2.2) is represented using a DOEM database. Finally, we discuss some properties of DOEM databases that make them well-suited for representing changes in semistructured data.

Intuitively, annotations are tags attached to the nodes and arcs of an OEM graph that encode the history of basic change operations on those nodes and arcs. There is a one-to-one correspondence between annotations and the basic change operations. Thus, nodes and arcs may have the following annotations:

- $cre(t)$: the node was created at time t.
- $upd(t, ov)$: the node was updated at time t; ov is the old value.
- $add(t)$: the arc was added at time t.
- $rem(t)$: the arc was removed at time t.

The set of all possible node annotations is denoted by **node-annot**, and the set of all possible arc annotations is denoted by **arc-annot**.

Using the above definitions of node and arc annotations, we now define a DOEM database. In the following definition, the function $f_N(n)$ maps a node n to a set of annotations on that node and the function $f_A(a)$ maps an arc a to a set of annotations on that arc.

Definition 7.3.1 A DOEM *database* is a triple $D = (O, f_N, f_A)$, where $O = (N, A, v, r)$ is an OEM database, f_N maps each node in N to a finite subset of **node-annot**, and f_A maps each arc in A to a finite subset of **arc-annot**. □

7.3.1 DOEM Representation of an OEM History

Given an OEM database O and a history $H = (t_1, U_1), ..., (t_n, U_n)$ that is valid for O, we would like to construct the DOEM *database representing* O *and* H, denoted by $D(O, H)$. $D(O, H)$ is constructed inductively as follows. We start with a DOEM database D_0 that consists of the OEM database O with empty sets of annotations for the nodes and the arcs of O. Suppose D_{i-1} is the DOEM database representing O and $(t_1, U_1), ..., (t_{i-1}, U_{i-1})$, for some $1 \leq i \leq n$. The DOEM database D_i is constructed by considering the basic change operations in U_i. Since the history is valid, we can assume some ordering L_i of the operations in U_i (Definition 7.2.2). Starting with D_{i-1}, we process the operations in L_i in order. Whenever the value of an object is updated, in addition to performing the update we attach an *upd* annotation to the node. This annotation contains the timestamp t_i and the old value of the object. When a new object is created or an arc added, in addition to performing the modification, we attach a *cre* or *add* annotation with the timestamp t_i. When an existing arc is removed, we do not actually remove the arc from the graph; instead, we simply attach a *rem* annotation to the affected arc with the timestamp t_i. Observe that this representation directly stores the changes themselves, not the before and after images of the changes, and thus takes the *snapshot-delta* approach discussed in Chapter 2.

Example 7.3.1 Consider the history described in Example 7.2.3, which transforms the OEM database of Figure 7.1 to that of Figure 7.2. The corresponding DOEM database is shown in Figure 7.3. We see that the DOEM database contains several annotations, depicted as boxes in the figure. For example, the annotations with timestamp "1Jan97" correspond to the first set of updates. Note that the *cre*, *add*, and *rem* annotations contain only the timestamp, while the *upd* annotation also contains the old value of the updated node (10, in our example). Also note that the removed "parking" arc from the "Janta" restaurant object to the "Lytton lot 2" parking object

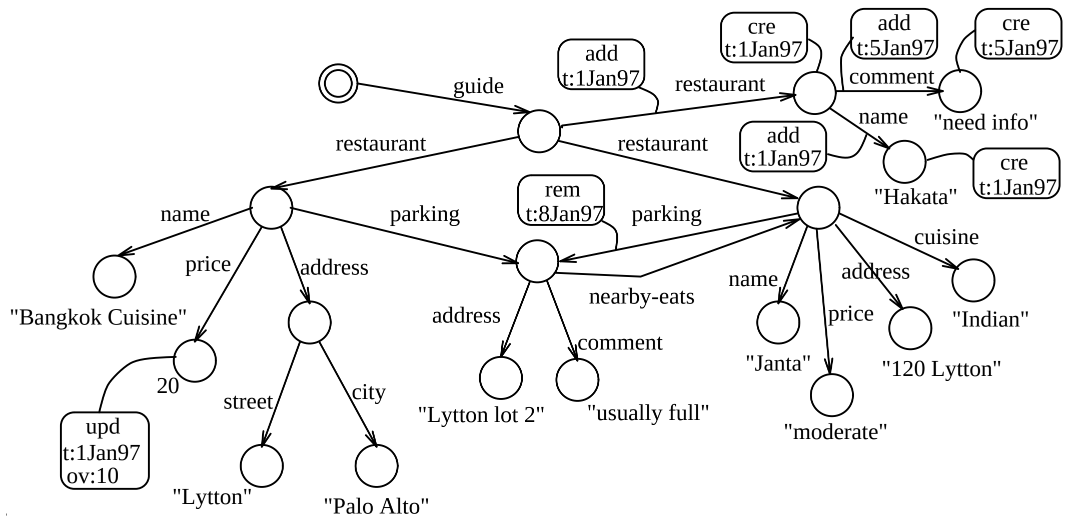


Figure 7.3: The DOEM object in Example 7.3.1.

is not actually removed from the DOEM database; instead it bears a *rem* annotation.
□

7.3.2 Properties of DOEM Databases

We have seen above how a DOEM database is used to represent an OEM database and its history. We now discuss the advantages of this representation. We say that a DOEM database D is *feasible* if there exists some OEM database O and valid history H such that $D = D(O, H)$. Note that we do not require DOEM databases to record all changes since creation, i.e., OEM database O need not be empty. DOEM databases have the following desirable properties:

- It is easy to obtain the *original snapshot* $O_0(D)$ from a DOEM database D. $O_0(D)$ contains exactly those nodes in D that do not have a *cre* annotation. The arcs of $O_0(D)$ are the arcs in D that either have no annotations, or have a *rem* annotation as the annotation with the smallest (earliest) timestamp.

- It is easy to obtain the *snapshot at time t*, $O_t(D)$, from a DOEM database D. Starting from the root object of D, we traverse D in preorder. For each node n

we encounter, we do the following:

1. We find the value $v_t(n)$ of n at time t (atomic value or $\mathcal{C}$) as follows: If n has no *upd* annotations, then $v_t(n) = v(n)$. Otherwise, let $upd(t_1, ov_1)$, ..., $upd(t_k, ov_k)$ be the *upd* annotations in $f_N(n)$. If $t_k \leq t$, $v_t(n) = v(n)$. Otherwise, pick $i \in [1, k]$ such that t_i is the smallest timestamp greater than t in $t_1, \ldots, t_k$; then $v_t(n) = ov_i$.

2. If $v_t(n) = \mathcal{C}$, continue the preorder traversal by following the arcs emanating from n that were present at time t. These are the arcs emanating from n that either do not have any annotation with timestamp less than or equal to t, or have an *add* annotation as the annotation with the greatest timestamp less than or equal to t.

- It is easy to obtain the *current snapshot* from a DOEM database. It is the *snapshot at time c*, where c is the current time.

- It is easy to obtain the *encoded history* $H(D)$ from a DOEM database D. The history $H(D) = (t_1, U_1), ..., (t_n, U_n)$ is constructed as follows. First, $t_1, \ldots, t_n$ is the set of timestamps occuring in D, ordered by time. For each $i = 1 \ldots n$, U_i contains the following operations:

  1. $addArc(p, l, c)$ ($remArc(p, l, c)$), if the arc (p, l, c) has the annotation $add(t_i)$ (respectively, $rem(t_i)$);

  2. $updNode(n, v)$, if n has an annotation $upd(t_i, ov)$ and v is the next value of n. That is, $v = ov'$ if the next (by time) annotation of n is $upd(t_j, ov')$, and $v = v(n)$ if n is not updated after t_i;

  3. $creNode(n, v)$, if n has the annotation $cre(t_i)$, where v is defined as in Case 2.

- It is relatively easy to determine if a given DOEM database D is feasible. We construct the original snapshot $O_0(D)$ and the encoded history $H(D)$ for D as above, and test if $D(O_0(D), H(D)) = D$.

- Most importantly, if D is feasible, we can show that the OEM database $O_0(D)$ and the history $H(D)$ encoded by D are unique. Thus, a DOEM database faithfully captures all the information about the history of the corresponding OEM database.

- As we will see in the next section, it is easy and intuitive to query the history encoded in a DOEM database.

7.4 Querying Over Changes

In Section 7.3, we have seen how the history of an OEM database is represented by the corresponding DOEM database. In this section, we describe how DOEM databases are queried. We introduce a query language called *Chorel* for this purpose. Chorel is similar to the Lorel language [AQM+96] used to query OEM databases. We begin with a brief overview of Lorel, followed by a description of the syntax and semantics of Chorel.

7.4.1 Lorel Overview

Lorel uses the familiar `select-from-where` syntax, and can be thought of as an extension of OQL [Clu98, Cat96] in two major ways. First, Lorel encourages the use of path expressions. For instance, one can use the path expression `guide.restaurant.address.street` to specify the streets of all addresses of restaurant entries in the Guide database. Second, in contrast to OQL, Lorel has a very "forgiving" type system. When faced with the task of comparing different types, Lorel first tries to coerce them to a common type. When such coercions fail, the comparison simply returns false instead of raising an error. This behavior, while it may be unsuitable for traditional databases, is exactly what a user expects when querying semistructured data. Lorel also provides a number of syntactic conveniences such as the possibility of omitting the `from` clause. We do not describe Lorel in detail here (see [AQM+96]), but only present through a simple example those features that are needed to understand Chorel.

Example 7.4.1 Consider again the OEM database depicted in Figure 7.2. To find all restaurants that have a price rating of less than 20.5, we can use the following Lorel query:

```
select guide.restaurant
where guide.restaurant.price < 20.5;
```

Note that the query expresses the price rating as a real number whereas the restaurant entries for "Bangkok Cuisine" and "Janta" in the OEM database shown in Figure 7.2 use an integer and a string, respectively. Furthermore, the third restaurant entry does not have a price subobject at all. Lorel successfully coerces the integer price 10 to real, and the comparison succeeds. For the string encoding of the price ("moderate"), Lorel tries to coerce, but fails, returning false as the result of the comparison. Finally, for the third restaurant, the missing price subobject simply causes the comparison to return false. Thus, the result of the above query is a singleton set containing the restaurant object for "Bangkok Cuisine." Note that this result is an intuitively reasonable response to the original query, despite the typing difficulties and the missing data. □

Lorel also allows the use of path expressions that include regular expressions and wildcards (e.g., "#" matches an arbitrary path of length 0 or more). Such *general path expressions* are powerful extensions of the simple path expressions of OQL, and allow Lorel users to specify complex path patterns in a database graph. Chorel is also based on extending the notion of path expressions, but in a different direction: We extend path expressions to allow the annotations in DOEM databases to be specified and matched.

7.4.2 Chorel

In Chorel, path expressions may contain *annotation expressions*, which allow us to refer to the node and arc annotations in a DOEM database. Informally, Lorel path expressions can be thought of as being matched to paths in the OEM database during query execution. Analogously, the annotation expressions in Chorel path expressions can be thought of as being matched to annotations on the corresponding paths in the DOEM database.

Example 7.4.2 Consider the DOEM database depicted in Figure 7.3. To find all newly added restaurant entries only, we can use the following Chorel query:

```
select guide.<add>restaurant;
```

The annotation expression "`<add>`" specifies that only those objects connected to the "guide" object by a "restaurant"-labeled arc having an *add* annotation should be retrieved. For the database depicted in Figure 7.3, this Chorel query returns the restaurant object with name "Hakata." □

Not surprisingly, we use four kinds of annotation expressions in Chorel path expressions: *node annotation expressions* "`cre`" and "`upd`," and *arc annotation expressions* "`add`" and "`rem`." Recall that a path expression, e.g., `guide.restaurant.price`, consists of a sequence of labels. Arc annotation expressions must occur immediately before a label, whereas node annotation expressions must occur immediately after one. (Note that since node and arc annotations use different keywords, no confusion can arise.) Path expressions containing node or arc annotation expressions are called *annotated path expressions*. For instance,

```
guide.<add>restaurant.price<upd>
```

is a correct annotated path expression. It requires an *add* annotation to be present on the arc labeled "restaurant," and an *upd* annotation on the "price" node (i.e., on the node at the destination of the arc labeled "price"). For simplicity, in this chapter we do not consider path expressions that have annotation expressions attached to wildcards or regular expressions, however generalizing to allow such annotation expressions is not difficult.

Annotation expressions may also introduce *time variables* to refer to the timestamps stored in matching annotations, and *data variables* to refer to the modified values in matching *upd* annotations. More precisely, the syntax of annotation expressions is as follows:

$<$*Annot* [`at` *timeV*]$>$	if *Annot* is in { `add`, `rem`, `cre` }
$<$ `upd` [`at` *timeV*] [`from` *oldV*] [`to` *newV*]$>$	for `upd`

where *timeV*, *oldV*, and *newV* are variables. Note that a DOEM database does not explicitly store the new value of an updated object, however this information is available implicitly, and can be determined easily as shown in Section 7.3.2.

Let us consider a Chorel query that uses a time variable. Note that we allow users to enter timestamps using a textual representation, e.g., 4Jan97. In keeping with Lorel's extensive use of coercion, any recognizable format is allowed and is converted automatically to an internal timestamp datatype.

Example 7.4.3 Consider the DOEM database in Figure 7.3. To find all restaurant entries that were added before January 4th, 1997, we can use the following Chorel query:

```
select guide.<add at T>restaurant
where T < 4Jan97;
```

The Chorel preprocessor will rewrite this query to obtain the following. (We will explain this rewriting shortly.)

```
select R
from guide.<add at T>restaurant R
where T < 4Jan97;
```

The introduced *from* clause will bind R to all "restaurant" objects that are connected to the "guide" object via an arc with an *add* annotation, and will provide corresponding bindings for T. More precisely, the evaluation of the `from` clause will yield the set of pairs $\langle R, T\rangle$ such that there is a `restaurant` arc from the `guide` object to R that has an *add* annotation with timestamp T. The `where` clause will filter out the $\langle R, T\rangle$ pairs for which T does not satisfy the condition. For the DOEM database in Figure 7.3, this query returns the restaurant object for "Hakata." □

Once time and data variables have been bound using annotations, they can be used just like other variables in Lorel or OQL. This feature is illustrated by the following query, which uses time and data variables in the `select` clause.

Example 7.4.4 Referring again to the DOEM database in Figure 7.3, suppose we want to find the names of all restaurants whose price ratings were updated on or after January 1st, 1997 to a value greater than 15, together with the time of the update and the new price. We can use the following query:

```
select N, T, NV
     from guide.restaurant.price<upd at T to NV>
     guide.restaurant.name N
     where T >= 1Jan97 and NV > 15;
```

The result of the above query is a single complex object with three components, as shown below. The label *name* is chosen by Chorel using the method described in [AQM$^+$96]. For time and data variables whose labels are not specified by the query, Chorel chooses the default labels *create-time, add-time, remove-time, update-time, new-value*, and *old-value*.

```
answer
     name "Bangkok Cuisine"
     update-time 1Jan97
     new-value 20
```

□

7.4.3 Chorel Semantics

We now make the semantics of Chorel queries more precise. As is done for Lorel, the semantics is described by specifying the rewriting of Chorel queries into OQL-like queries. However, we need to introduce some additional machinery to handle the annotation expressions in Chorel queries.

First, the annotation expressions in a Chorel query are transformed into a canonical form that includes all variables. For example, "`<add>`" is rewritten to "`<add at`

`T1>`," and "`<upd from X>`" is rewritten to "`<upd at T2 from X to NV2>`," where `T1`, `T2`, and `NV2` are fresh variables. Next, as in Lorel, we eliminate path expressions by introducing variables for the objects "inside" the path expressions. For example, the path expression "`a.b.c`" in a `from` clause is converted to "`a.b X, X.c Y`," where `X` and `Y` are new *range variables*. The details of this rewriting are described in [AQM+96].

At this stage, we have to give a semantics to range variable definitions that may include annotation expressions (e.g., "`X.label Y`," "`X.<add at T>label Y`") in the context of a DOEM database. In the absence of an annotation expression, the semantics of an expression "`X.label Y`" is that for a binding o_X of X, Y is bound to all objects o_Y such that there is an arc labeled *label* from o_X to o_Y in the current snapshot. Note that by this semantics, a standard Lorel query (without annotations) over a DOEM database has exactly the semantics of the same query asked over the current snapshot for that DOEM database. In the presence of annotation expressions, the semantics requires the existence of the specified annotation, and also provides bindings for the variables in the annotation expression. The bindings are also specified by a special rewriting. As an example, the query in Example 7.4.4 is rewritten to:

```
select N, T, NV
from guide.restaurant R, R.price P, R.name N,
     (T, OV, NV) in updFun(P)
where T >= 1Jan97 and NV > 15;
```

Our rewriting uses the following functions, which extract the information stored in annotations:

$$\begin{aligned}
&\mathit{creFun}(\mathit{node}) \rightarrow \{\mathit{time}\}\\
&\mathit{updFun}(\mathit{node}) \rightarrow \{(\mathit{time}, \mathit{old\text{-}value}, \mathit{new\text{-}value})\}\\
&\mathit{addFun}(\mathit{source}, \mathit{label}) \rightarrow \{(\mathit{time}, \mathit{target})\}\\
&\mathit{remFun}(\mathit{source}, \mathit{label}) \rightarrow \{(\mathit{time}, \mathit{target})\}
\end{aligned}$$

The function *creFun(n)* returns the set of timestamps found in *cre* annotations on node n. (Note that by our definition of change operations in Section 7.2.1, this set is either empty or a singleton.) The function *updFun(n)* returns a set of triples corresponding to the timestamp, the old value, and the new value in *upd* annotations on n. The function *addFun(n,l)* returns a set of (t, c) pairs such that c is an l-labeled subobject of n via an arc that has an $add(t)$ annotation. The *remFun* function is analogous to *addFun*. Once this rewriting has been performed, the `from`, `where`, and `select` clauses of the resulting query are processed in a standard manner.

Above, we have illustrated how variables introduced in the `from` clause are interpreted. Variables may be introduced in the `where` clause as well. They are treated by introducing existential quantification in the `where` clause, extending the treatment of such variables in Lorel [AQM+96]. Consider the following example:

Example 7.4.5 Consider again the DOEM database of Figure 7.3. Suppose we want the names of restaurants to which a "moderate" price subobject was added since January 1st, 1997. We can write the following Chorel query:

```
select N
from guide.restaurant R, R.name N
where R.<add at T>price = "moderate" and T >= 1Jan97;
```

The variable `T` is introduced in the `where` clause. Therefore, the rewritten `where` clause is:

```
where exists (T, P) in addFun(R,"price") :
     (P = "moderate" and T >= 1Jan97);
```

□

7.5 Implementing DOEM and Chorel

In this section, we describe how we have implemented DOEM databases and Chorel queries. One approach would be to extend the kernel of the *Lore* database system [MAG+97] to allow annotations to be attached to the nodes and arcs of an OEM database. Given these extensions, the Lorel query engine could be extended to a

Chorel query engine in a straightforward manner based on the semantics specified in Section 7.4.3. We do not discuss this approach further. Instead, our implementation uses an alternative approach of implementing DOEM and Chorel "on top of" Lore. We encode DOEM databases as OEM databases, and we implement Chorel by translating Chorel queries to equivalent Lorel queries over the OEM encoding of the DOEM database. In addition to being more modular than the direct implementation approach (and not affecting Lore object layout or query processing), this approach can also be adapted easily to other graph-based data models, e.g., those in [BDHS96, Cat96]. Note that while there are several simple methods of encoding a DOEM database as an OEM database, the challenge here is to devise an encoding that permits a simple and valid translation of Chorel queries over the original DOEM database into Lorel queries over the OEM encoding. For many of the obvious possible encodings, such query translation proves to be very difficult or impossible.

We begin by explaining how we encode DOEM databases in OEM, followed by a description of the translation of Chorel queries to Lorel queries for this encoding, and finally a description of our system implementation.

7.5.1 Encoding DOEM in OEM

Let D be a DOEM database. We encode D as an OEM database O_D defined as follows. For each object o in D, there is a corresponding object o' in O_D. Atomic objects are encoded as complex objects so that we can record their histories using subobjects. Special labels used by the encoding start with the character "`&`" to distinguish them from standard labels occuring in O. The encoding object o' for DOEM object o has the following subobjects, listed by their labels. Refer to Figures 7.4 and 7.5.

- `&val`: If o is atomic with current value v, there is a "`&val`"-labeled arc from o' to an atomic object with value v. If o is complex, there is a "`&val`"-labeled arc from o' to itself. (The use of this extra edge will soon become clear.)

- `&cre`: If o has a create annotation $cre(t)$, then o' has a "`&cre`"-labeled complex subobject o'_c that has a "`&time`"-labeled atomic subobject with value t.

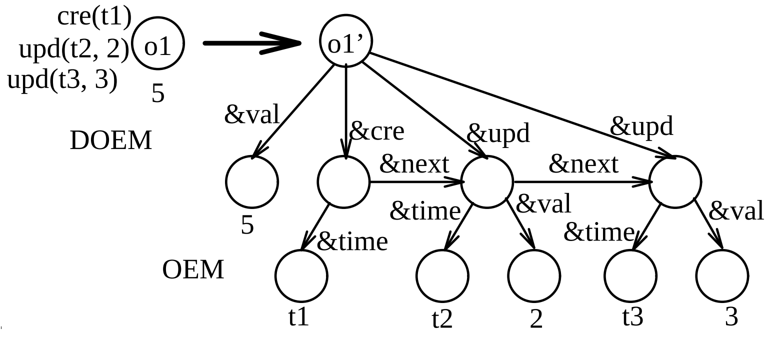


Figure 7.4: Encoding a DOEM object in OEM: node annotations

- `&upd`: For each update annotation $upd(t, ov)$ attached to o, o' has an "`&upd`"-labeled complex subobject o'_u. The object o'_u has a "`&time`"-labeled atomic subobject with value t, and a "`&val`"-labeled atomic subobject with the value before the update (ov).

- l: If the current snapshot for D contains an arc (o, l, p), then O_D contains an arc labeled l from o' to the object p' that encodes p.

- `&`l`-history`: If D contains an arc (o, l, p), then O_D contains an arc $(o', \&l\texttt{-history}, o'_l)$ where o'_l is a complex object that contains the history of the l arcs from o to p. The object o'_l has the following structure:

  - `&target`: There is an arc $(o_l, \&\texttt{target}, p')$, where p' is the object encoding p.

  - `&add`, `&rem`: For each annotation $add(t)$ ($rem(t)$) attached to (o, l, p), there is an "`&add`"-labeled (respectively, "`&rem`"-labeled) complex subobject o'_c that has a "`&time`"-labeled atomic subobject with value t.

- `&next`: For each OEM object o'_1 that encodes a DOEM object o_1 and its node annotations, the "`&cre`"- and "`&upd`"-labeled subobjects of o'_1 are chained together in ascending order of the values of their "`&time`" subobjects using arcs with label "`&next`." (As we shall see shortly, this chaining is useful for obtaining

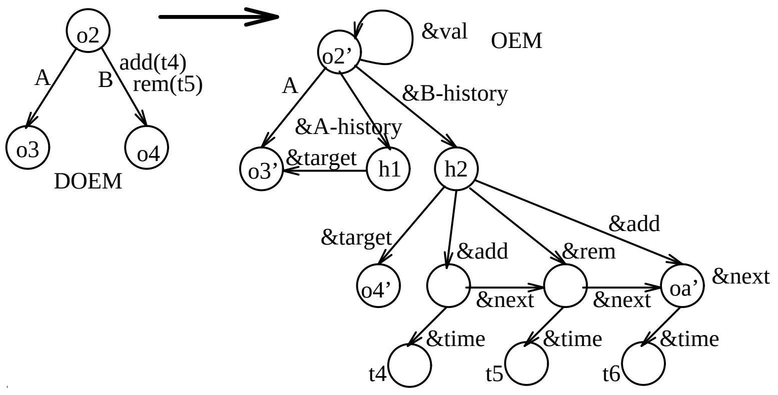


Figure 7.5: Encoding a DOEM object in OEM: arc annotations

the "new value" corresponding to an update annotation.) Similarly, for each OEM object o'_{iLj} that encodes a DOEM arc (o_i, L, o_j) and the annotations on that arc, the "`&add`"- and "`&rem`"-labeled subobjects of o_{iLj} are chained together in ascending order of the values of their "`&time`" subobjects using arcs with label "`&next`." (As we shall see in Section 7.6, this chaining is useful for implementing snapshot-based access.)

7.5.2 Translating Chorel to Lorel

Given the above encoding of a DOEM database as an OEM database, we now describe how a Chorel query over a (conceptual) DOEM database is translated into an equivalent Lorel query over an OEM encoding of the DOEM database. In Section 7.4.3 we described how a Chorel query can be rewritten into an OQL-like query using special functions *creFun*, *updFun*, *addFun*, and *remFun*. Therefore, in the following we assume that we are given such a rewritten query.

We simulate the special functions *creFun*, *updFun*, *addFun*, and *remFun* using expressions that extract the required values from the OEM encoding of the annotations. For example, the expression "`(T, OV, NV) in` *updFun*`(P)`" is replaced with "`P.&upd U, U.&time T, U.&val OV, U.&next.&val NV.`" From the encoding scheme described in Section 7.5.1, we see that this expression instantiates the triple `(T, OV,` `NV)` to the timestamp, old value, and new value of the update annotations on objects bound to `P`. If an expression of the form "`(T, C) in` *addFun*`(P, l)`" occurs in a Chorel query, we replace it with "`P.&l-history H, H.&add.&time T, H.&target C`." The case for remove annotations, involving the *remFun* function, is analogous. Finally, we replace an expression "`T in` *creFun*`(P)`," with "`P.&cre.&time T`."

Note that our encoding scheme ensures that only arcs that exist in the current snapshot corresponding to the encoded DOEM database are accessible directly via their labels in the encoding. If an l-labeled arc does not exist in the current snapshot, its information is stored using an arc with label `&`l`-history`, which does not match the label l.

One remaining issue is that in the OEM encoding of a DOEM database, the value of an atomic object is stored in a "`&val`"-labeled subobject of the encoding object. So, for instance, when a query compares an atomic object to a value, we want to use the value stored in the "`&val`" subobject for this comparison. Therefore, wherever in the query the value of a object variable is accessed (i.e., in predicates and function arguments) we replace the object variable "`X`" with "`X.&val`." Observe that since there is a "`&val`"-labeled arc from the encoding of each complex object to itself, we can safely perform the above transformation for all value accesses of object variables occuring in the original query, without knowing whether the objects they encode are atomic or complex (which, in general, we will not know). The transformation is illustrated by the following example.

Example 7.5.1 Consider the Chorel query in Example 7.4.5. In Section 7.4.3, we considered the OQL-like rewriting of this query. We now complete this rewriting as described above, to yield the following Lorel query over the OEM encoding of the DOEM database in Figure 7.3:

```
select N
from guide.restaurant R, R.name N
where exists H in R.&price-history :
      exists P in H.&target :
      exists T in H.&add.&time :
            T >= 1Jan97 and P.&val = "moderate";
```

Note that we simulate the range specification *addFun(R,"price")* using the "&"-prefixed subobjects. Further, we use *P.&val* to access the actual price value (and not the complex object packaging it with its history). □

Note that the example query returns a set of DOEM objects that represent restaurant names. That is, it returns not only the names of the restaurants, but also the history of these names, if they changed. Returning the DOEM object enables the user to access both the value and the history of an object.

In the above description, for simplicity we assumed that every atomic object o is encoded using a complex object o' that has a `&val`-labeled subobject with value $v(o)$. However, in practice we do not encode unannotated atomic objects in this manner; that is, if an atomic object o has no annotations, we encode it using a simple atomic object o' with value $v(o)$. In our translation scheme, we replace accesses to the value of an variable `X` by `X.[&val]`, which is a Lorel path expression indicating an optional path component `&val`.

7.5.3 Implementation

Figure 7.6 depicts the system architecture of CORE, a Change Object Repository based on DOEM and Chorel.

A DOEM database is first populated by loading a *DOEM load file*, which is a simple textual representation of a DOEM database. The *Encoder* reads this DOEM load file and produces a Lore load file that encodes the DOEM database using the method described in Section 7.5.1. The Lore loader reads the OEM load file and stores the corresponding database in Lore [MAG+97].

When a user invokes a Chorel query on the DOEM database, the query is first translated into a Lorel query over the OEM encoding of the DOEM database by the *Chorel Translator*, using the method described in Section 7.5.2. The resulting Lorel query is evaluated over the OEM database by the Lore query engine. Note that the result of the Lorel query contains OEM objects that are encodings of DOEM objects and annotations. The *API (Application Program Interface) Translator* translates these OEM encodings to the corresponding DOEM objects, which can then be displayed by

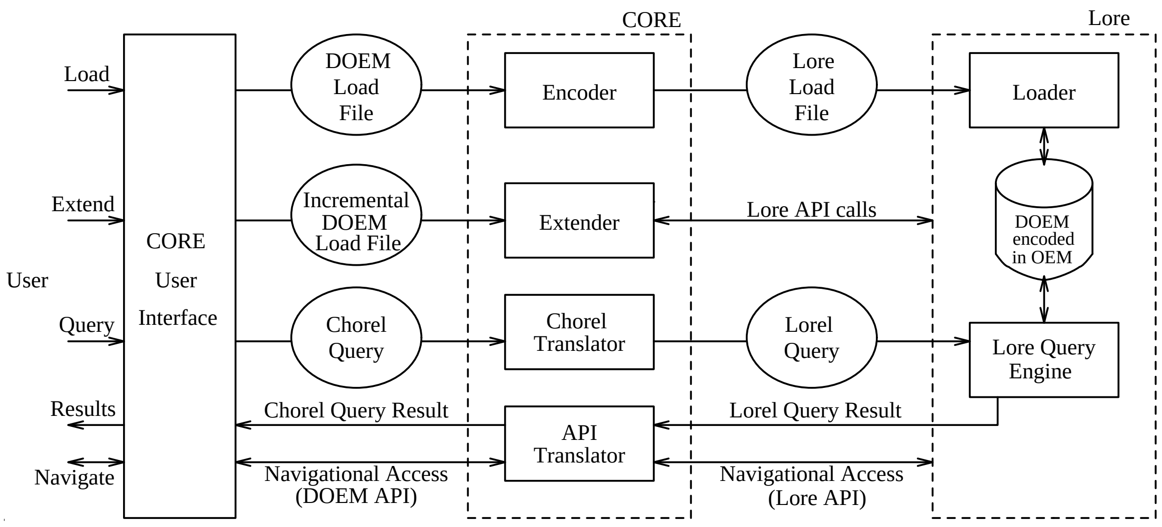


Figure 7.6: System architecture

the *User Interface*.

The User Interface can also be used to browse the DOEM database, either directly, or starting from the results of a Chorel query. The translation from navigation in the DOEM database to navigation in the OEM encoding stored in Lore is done by the API Translator.

A DOEM database can also be *extended* by adding new data and changes. For example, consider a DOEM database representing the history of our Guide database (Example 7.2.1) up to last week. We may want to extend the database to include this week's changes when they become available. This capability is handled by the *Extender*, which takes as input an *incremental* DOEM *load file*, and uses the Lore API to modify the encoded DOEM database. We are also in the process of extending Lore so that it can monitor changes to OEM databases and create and extend corresponding DOEM databases directly.

7.6 Virtual Annotations and Snapshot-based Access

In Section 7.4.2 we have seen how the construct `<upd at T from oldV to newV>` refers to a *virtual annotation* $upd(t, ov, nv)$, where t, ov, and nv are, respectively, the timestamp, the old value, and the new value of an update operation in the history. The real annotation, $upd(t, ov)$, does not contain the old value, however that information is available elsewhere in the database. We can extend this idea of virtual annotations to facilitate access to other implicit information in a DOEM database. As a concrete example, in this section we introduce virtual annotations that facilitate *snapshot-based* access to a DOEM database. We define the semantics of Chorel queries containing references to virtual annotations by using range functions that are defined over the real annotations and data in a DOEM database. We describe how to implement this added functionality by extending the translation-based method of Section 7.5.

7.6.1 Snapshot-based Access

Recall from Section 7.4.3 that an unannotated path expression such as `guide.restaurant.entree.price` is evaluated over the *current snapshot* of a DOEM database. Sometimes, one may wish to evaluate path expression components over other (non-current) snapshots. For example, we may wish to refer to the price of an entree at some time T; we introduce the syntax `guide.restaurant.entree.price<at T>`. Similarly, we may wish to refer to the existence of a *parking* arc between two objects X and Y at time T; we use the syntax `X.<at T>parking Y` in the `from` clause of a Chorel query.

Example 7.6.1 Consider the Guide database depicted in Figure 7.3. Suppose we wish to list the parking areas close to the restaurant "Janta" as of 1st January 1997. We write the following query:

```
select P
from guide.restaurant R, R.<at T>parking P
where R.name = "Janta" and T = 1Jan97;
```

For the DOEM database depicted in Figure 7.3, this query returns the parking object with address "Lytton lot 2," since on 1st January 1997 there was a "parking" arc from the Janta restaurant object to the Lytton parking object. (This arc was removed on 8th January 1997.) □

When the variable T occuring in an `at` annotation expression is bound to a constant elsewhere in the query (as in the above example), the effect of the annotation expression on query evaluation is intuitively simple: We evaluate the query as if the path expression component qualified by `<at T>` refers to the snapshot of the database at time T. As we have seen in Section 7.3.2, the snapshot at time T is easily obtained using the information in a DOEM database. However, if T is unbound, then unless we take special precautions we may find ourselves faced with unsafe queries, as illustrated by the following example.

Example 7.6.2 For the Guide database depicted in Figure 7.3, suppose we are interested in finding the times at which the restaurant "Bangkok Cuisine" had a price rating less than 15. We write the query as follows:

```
select T
from guide.restaurant R, R.price<at T> P
where R.name = "Bangkok Cuisine" and P < 15;
```

The basic problem with this query is that while the database stores only a finite number of timestamps, the above query would require T to range over the infinite number of intermediate timestamp values as well. □

We overcome such difficulties by allowing timestamp variables such as T above to bind only to those timestamp values that exist explicitly in the DOEM database. Intermediate timestamp values are represented using intervals $[B, E)$, where B and E are the begin and end timestamps, respectively. (We use a convention of intervals that are closed on the left and open on the right; our methods are not dependent on this convention.)

To introduce this concept of intervals, we add another virtual annotation, called *during*, on nodes and arcs, and a corresponding annotation expression "`<during B`

`E>`" in the syntax of annotated path expressions. (As we will see in Section 7.6.3, virtual annotation *during* in fact subsumes virtual annotation *at*.) Intuitively, the construct `X<during B E> V` in a `from` clause binds the triple (B, E, V) to all values $\{(b, e, v)\}$ such that the object X had value v continuously from time b to time e. Similarly, the construct `X<during B E>l Y` binds the triple (B, E, Y) to all values $\{(b, e, Y)\}$ such that the arc (X, l, Y) existed continuously from time b to time e. We further require that the above intervals $[b, e)$ be maximal.

When using snapshot-based access, we often need to refer to the current time. We introduce a distinguished timestamp t_N for this purpose. More precisely, t_N is a special variable whose value during the evaluation of a query is the time at which that evaluation begins. Similarly, we often need to refer to the initial timestamp corresponding to a database; we introduce a distinguished timestamp t_I for this purpose. More precisely, each DOEM database has an initial timestamp t_I associated with it. Note that t_I is a constant, and may be negative infinity.

Using the `during` virtual annotation, the query in Example 7.6.2 may be rewritten as follows:

```
select B,E
from guide.restaurant R, R.price<during B E> P
where R.name = "Bangkok Cuisine" and and P < 15;
```

This query returns a set of pairs $\{(b, e)\}$ such that at all times during the interval $[b, e)$, Bangkok Cuisine had a price rating less than 15. For our example database depicted in Figure 7.3, this query returns the singleton set $\{(t_I, 1Jan97)\}$, where t_I is the initial timestamp of the database.

Note that it *is* possible to express such snapshot-based queries using only the basic Chorel constructs described in Section 7.4. However, the resulting queries are extremely cumbersome. For example, the simple snapshot-based access `X.<during B E>foo Y` in a *from* clause requires a construction such as the following:

```
from X.<add at B>foo Y, X.<rem at E>foo Z...
where Y = Z and not exists M :
        (X.<add at M>foo Y or X.<rem at M>foo Y);
```

In reality, the expression is even more complex, since we need to handle the special cases involving missing annotations on both the "begin" and the "end" side. Thus, snapshot-based access is an excellent candidate for simplification using virtual annotations.

7.6.2 Semantics of `during`

We now formalize our intuitive description of the semantics of `during` annotations. As in Section 7.4.3, we shall specify the semantics using a rewriting with special functions for binding range variables. To define the semantics of the arc annotation expression `X.<during B E>l Y` in the `from` clause of a Chorel query, we introduce a special function, *arcDuring*. This function maps a DOEM object o_1 and label l to a set of triples $\{(b, e, o_2)\}$ such that in the history represented by the DOEM database, the arc (o_1, l, o_2) existed in the time interval $[b, e)$, and $[b, e)$ is maximal (i.e., this condition fails to hold if we decrease b or increase e). We rewrite the `from` clause by replacing `X.<during B E>l Y` with `(B,E,Y) in` *arcDuring*`(X,l)`. (Recall from Section 7.3.2 that given a DOEM database D, it is easy to obtain the snapshot at time t, $O_t(D)$. Thus the intervals $[b, e)$ in the definition of *arcDuring* are well defined.) The function *arcDuring* has some notable boundary cases: If the earliest annotation on an arc is $rem(t_1)$, then the arc exists in $[t_I, t_1)$. (Recall from Section 7.6.1 that t_I is the initial timestamp associated with a database and t_N is the current timestamp.) Similarly, if the latest annotation on an arc is $add(t_2)$, then the arc exists in the interval $[t_2, t_N]$. Finally, if an arc has no annotations, it exists in $[t_I, t_N]$.

Now we define the semantics of the node annotation expression `X<during B E> V` in the `from` clause of a Chorel query. To do so, we introduce a special function, *nodeDuring*. This function maps a DOEM object o to a set of triples $\{(b, e, v)\}$ such that in the history represented by the DOEM database, the object o had value v during the time interval $[b, e)$, and $[b, e)$ is maximal. We rewrite the `from` clause replacing `X<during B E> V` with `(B,E,V) in` *nodeDuring*`(X)`. (Using Section 7.3.2 we see that the intervals $[b, e)$, and the corresponding values v, are well-defined.) The function *nodeDuring* also has some notable boundary cases: If the earliest annotation on a

DOEM object o is $upd(t_1, v_1)$ then o has value v_1 in the interval $[t_I, t_1)$. Similarly, if the latest annotation on o is $upd(t_k, v_k)$, then o has value $v(o)$ (the current value) in $[t_k, t_N]$. Finally, if o has no annotations, then it has value $v(o)$ in $[t_I, t_N]$.

Example 7.6.3 Consider the query proposed in Example 7.6.1. Using the `during` construct, we can write the following query to return parking for the "Janta" restaurant as of 1st January 1997.

```
select P
from guide.restaurant R, R.<during B E>parking P
where R.name = "Janta" and B <= 1Jan97 and E > 1Jan97;
```

Using the semantics for `during` described above, we see that this query is conceptually rewritten to the following:

```
select P
from guide.restaurant R, (B,E,P) in arcDuring(R,parking)
where R.name = "Janta" and B <= 1Jan97 and E > 1Jan97;
```

Consider the DOEM database in Figure 7.3. When R is bound to the restaurant object "Janta," function *arcDuring* results in the tuple variable (B, E, P) ranging over the singleton set $\{(t_I, 8Jan97, p_1)\}$, where p_1 is the parking object with address "Lytton lot 2." Since R, B, and E satisfy the predicate in the `where` clause, the Lytton parking object will be returned as the query result. □

7.6.3 The `at` Construct

Examples 7.6.1 and 7.6.3 suggest a simple definition for the edge annotation `X.<at T>l Y` and the node annotation `X<at T> V`. We define them as abbreviations for `X.<during B E>l Y` and `X<during B E> V`, respectively, and add the condition `B <= T < E` to the `where` clause. Note that our rewriting requires the variable `T` occuring in the `at` annotation to be bound elsewhere in the query independently of the path expression component containing `at`. For example, if we apply this definition of `<at T>` to rewrite the query in Example 7.6.1, we obtain the query in Example 7.6.3.

In cases where the variable `T` occuring in the `<at T>` construct is not bound elsewhere in the query, the definition of `at` as an abbreviation for a `during` expression fails. For example, if we apply the rewriting to the problematic query of Example 7.6.2, which uses `<at T>` without binding `T` elsewhere, we get the following query in which `T` is still unbound:

```
select T
from guide.restaurant R, R.price<during B E> P
where R.name = "Bangkok Cuisine" and P < 15 and B <= T and T < E;
```

In general, this problem can be mitigated by allowing timestamp variables such as `T` to bind to intervals instead of single timestamps. However, we do not consider such extensions further in this chapter. We shall henceforth assume that the `<at T>` construct is defined only when `T` is bound elsewhere in the query independently of the path expression component containing `at`.

7.6.4 The `snap` Construct

Let us now consider a special class of Chorel queries that are useful in studying past states of a historical database. Intuitively, such queries take the snapshot at some time t, and then evaluate an ordinary (non-historical) query over this snapshot. We call such queries *pure snapshot queries*. For example, using our Guide database, suppose we wish to generate, as of 15th June 1997, the names, price ratings, and parking addresses for restaurants with a price rating less than 20. That is, we would like to evaluate the following Lorel (non-historical) query over the OEM database that is the DOEM snapshot of 15th June 1997:

```
select R, P, A
from guide.restaurant R, R.price P, R.parking.address A
where R.price < 20;
```

In reality we are evaluating Chorel queries over our DOEM database. Thus, to express that the above query should be evaluated over the snapshot of 15th June 1997, we could qualify each component of each path expression in the query as follows:

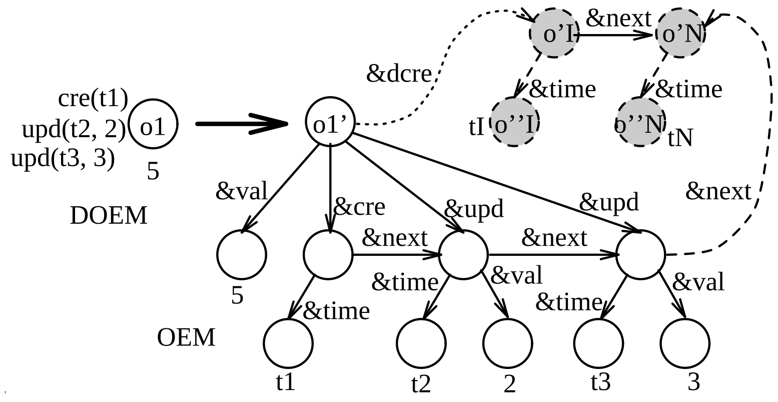


Figure 7.7: Encoding a DOEM object in OEM: node annotations

```
select R, P, A
from guide.<at T>restaurant R, R.<at T>price<at T> P,
     R.<at T>parking.<at T>address<at T> A
where R.<at T>price<at T> < 20 and T = 15Jun97;
```

In order to make writing such *snapshot queries* more convenient, we introduce as a syntactic convenience the construct `<snap T>`, with the requirement that `T` be bound elsewhere in the query independently of the path expression component containing `snap`. The construct `X.<snap T>foo Y` in a `from` clause is rewritten to `X.<at T>foo Y`; furthermore, any other use of `Y` in the query is (recursively) rewritten as though it were qualified by a `<snap T>`. In particular, `Y.bar Z` is interpreted as `Y.<snap T>bar Z` and recursively rewritten, and accesses to `Y`'s value are rewritten as `Y<at T>`. The `where` clause is handled analogously. Using this construct, the above query may now be written more simply as follows:

```
select R, P, A
from guide.<snap T>restaurant R, R.price P, R.parking.address A
where R.price < 20 and T = 15Jun97;
```

7.6.5 Implementing `during` by translation

We now describe how the translation-based implementation of Chorel described in Section 7.5 is extended to accommodate the `during` construct. Refer to Figures 7.7 and 7.8, which depict the OEM encoding of DOEM objects; we have indicated the new features using dashed lines. (The other features were described in Section 7.5.1.)

Each OEM database used to encode a DOEM database has a special complex object o'_N that has one "`&time`"-labeled atomic subobject o''_N with value t_N. (Recall, from Section 7.6.1, that t_N refers to the current time; in the implementation, the value of o''_N is the query execution time.) Similarly, there is a special complex object o'_I that has one "`&time`"-labeled atomic subobject o''_I with value t_I. (Recall, from Section 7.6.1, that t_I is the initial timestamp associated with a DOEM database, and may be negative infinity.) Note that there is exactly one instance of each of the objects o'_N, o''_N, o'_I, and o''_I per database. (To highlight this fact, these objects are depicted using shaded circles in Figures 7.4 and 7.5.)

In Section 7.5.1, we described the use of "`&next`"-labeled arcs to chain annotation-encoding objects in ascending order of the annotation timestamps. We now extend this chain to include the timestamps t_I and t_N as follows. Consider first the encoding of node annotations, as depicted in Figure 7.4. If a DOEM node o has one or more node annotations (create or update), then in its OEM encoding, we add a "`&next`"-labeled arc from the object encoding the annotation with the largest timestamp to the special object o'_N. The "`&next`"-labeled arc from o'_u to o'_N in Figure 7.4 is an example of this case. If the DOEM node o has no annotations, then in the OEM encoding, we add a "`&dcre`"-labeled arc from the corresponding node o' to the special node o'_I. In Figure 7.4, if o_1 were to not have a create annotation, a "`&dcre`"-labeled arc from o'_1 to o'_I would exist. (Since in reality o_1 does have a create annotation, this "`&dcre`"-labeled arc does not exist, and is depicted using a dotted line.)

Now consider the encoding of arc annotations, as depicted in Figure 7.5. If an arc (o_1, l, o_2) in the DOEM database has no annotations, then in the OEM encoding of the database, we add a "`&dadd`"-labeled arc from o'_{1l2} to the special object o'_I, where o'_{1l2} is the "`&`l`-history`"-labeled subobject of o'_1 that encodes the history of (o_1, l, o_2). In Figure 7.5, o_{1l2} is shown as the object h_1. If the arc (o_1, l, o_2) has one or more

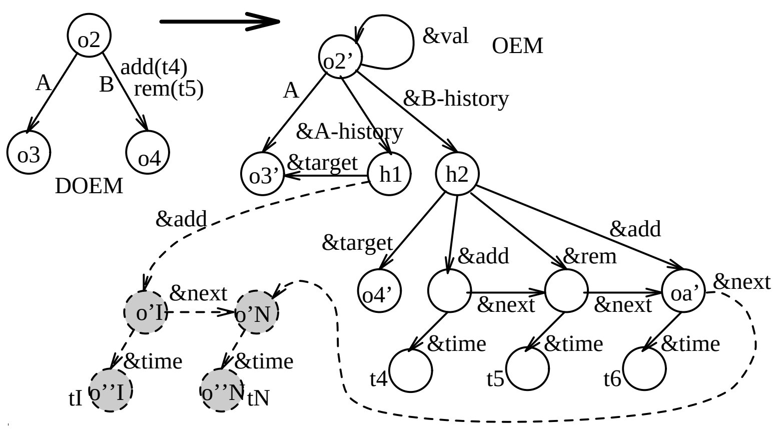


Figure 7.8: Encoding a DOEM object in OEM: arc annotations

annotations, and the annotation with the largest timestamp is an *add* annotation, then the OEM encoding has a "`&next`"-labeled arc from the corresponding "`&add`"-labeled subobject o'_a of o'_{1l2} to the special object o'_N. In Figure 7.5, we see an example of such an arc from o'_a to o'_N.

Given the above enhancements to our scheme for encoding DOEM in OEM, we can rewrite Chorel queries containing the `during` construct as Lorel queries over the encoding objects. Given a Chorel query with the construct `X<during B E>` in the `from` clause, we replace this construct by the following: `X(.&cre|.&upd|.&dcre) A, A.&time B, A.&next.&time E, A.&next.&val V`. Similarly, if a Chorel query has the construct `X.<during B E>foo Y` in the `from` clause, we replace this construct by the following: `X.&foo-history H, H.&target Y, H(.&add|.&dadd) A, A.&time B, A.&next.&time E`. As in Section 7.5, variables introduced in the `where` clause of a Chorel query are treated by introducing existential quantification in the `where` clause.

Example 7.6.4 Consider the `during`-based query in Example 7.6.3. Using the above rewriting, we obtain the following Lorel query over the OEM database encoding the Guide DOEM database:

```
select P
```

```
from guide.restaurant R, R.&parking-history H, H.&target P,
     H.&add A, A.&time B, A.&next.&time E
where R.name = "Janta" and B <= 1Jan97 and E > 1Jan97;
```

□

7.6.6 Object Deletion and Garbage Collection

Recall that in the OEM data model underlying DOEM and Chorel, there is no explicit object deletion operation. Instead, persistence is by reachability from the distinguished root of the database, and any unreachable objects are implicitly deleted. An OEM database system must therefore periodically perform garbage collection in order to detect and remove such deleted objects. Between the time an object becomes unreachable and the time garbage collection is performed, the semantically deleted object continues to exist in the database. This situation does not pose any difficulties for Lorel queries, since Lorel path expressions cannot access any object that is unreachable from the root of the current database snapshot. However, in Chorel, such deleted objects are reachable using annotated path expressions that contain a "forward jump in time" (i.e., path expressions that refer to a more recent snapshot from an older one). The following example illustrates the point:

Example 7.6.5 Referring back to our Guide database depicted in Figure 7.3, suppose the arc from the Guide object to the restaurant object for "Bangkok Cuisine" is removed on 1st July 1997. This arc removal results in the restaurant object for Bangkok Cuisine, as well as its price, address, street, and city subobjects becoming unreachable from the root of the database, implying their deletion. In our DOEM database, however, these objects continue to exist; the only change is that there is now a remove annotation $rem(1Jul97)$ on the restaurant arc that was removed. Now suppose on 15th July 1997 we issue the following query to our DOEM database, asking for the current price rating of all restaurants that existed as of 1st June 1997:

```
select P
from guide.<at 1Jun97>restaurant R, R.price P;
```

Now since the *price* object for Bangkok Cuisine does not currently exist, the result of the above query should not contain it. However, there is no way for the Chorel query engine to detect this situation, since there is no information on either the *restaurant* or the *price* objects that suggests their deletion. (The relevant piece of information is the *rem* annotation on the restaurant arc.) Thus the query result will contain the price rating for Bangkok Cuisine. □

We mitigate the above problem by introducing a *delete annotation*, which records the deletion of an object (usually as a result of garbage collection). Suppose that at time t_G, some objects are determined to be newly unreachable from the root of the database. In the corresponding DOEM database, we mark such newly unreachable objects (which continue to exist physically) using a $del(t_G)$ annotation. We further ensure that we do not access the value of an object at time t' if that object has a $del(t)$ annotation with $t' > t$. More precisely, we modify the definition of the *nodeDuring* function in Section 7.6.2 to state that if a node has a $del(t_d)$ annotation then its value after t_d is undefined. (That is, the most recent time interval is modified from $[t_k, t_N]$ to $[t_k, t_d)$.) The corresponding changes to the translation-based implementation are straightforward.

7.7 A Query Subscription Service

In Section 7.1, we mentioned as an important application of change management being able to notify "subscribers" of changes in (semistructured) information sources of interest to them. In this section, we describe our design and implementation of such an application, called a *Query Subscription Service* (QSS), using DOEM and Chorel.

An ordinary query is evaluated over the current state of the database, the results are passed to the client and then discarded. An example of an ordinary query is "find all restaurants with Lytton in their address." In contrast, a *subscription query* is a query that repeatedly scans the database for new results based on some given criteria and returns the changes of interest. An example of a subscription query is "every week, notify me of all *new* restaurants with Lytton in their address." Below, we describe how subscription queries are specified and implemented in our system.

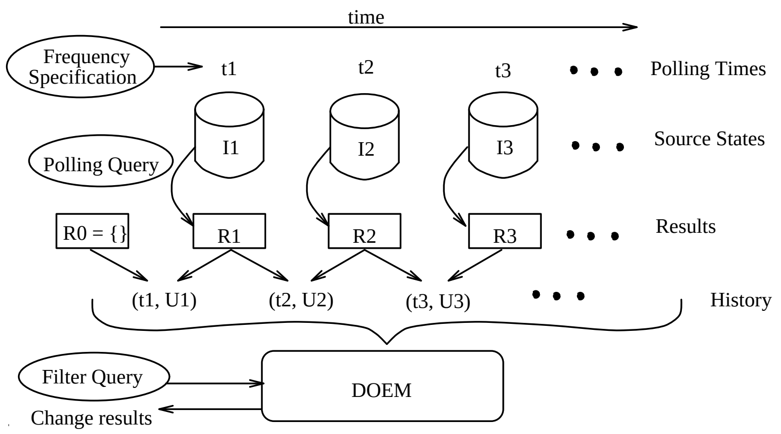


Figure 7.9: A Query Subscription Service based on DOEM and Chorel

Supporting subscription queries introduces the following challenges. First, as discussed earlier, many information sources that we are interested in (e.g., library information systems, Web sites, etc.) are *autonomous* [SL90] and typical database approaches based on triggering mechanisms are not usable. Second, these information sources typically do not keep track of historical information in a format that is accessible to the outside user. Thus, a subscription service based on changes must monitor and keep track of the changes on its own, and often must do so based only on sequences of snapshots of the database states.

Briefly, our approach to constructing a query subscription service over semistructured, possibly legacy, information sources, is as follows: We access the information sources using Tsimmis *wrappers* or *mediators* [PGGMU95, PGMU96], which present a uniform OEM view of one or more data sources. We obtain snapshots of relevant portions of the data and use differencing techniques from Chapters 4, 5, and 6 to infer changes based on these snapshots. Finally, we use DOEM to represent the changes, and Chorel to specify the changes of interest. We describe our approach in more detail next.

A *subscription* consists of three main components; refer to Figure 7.9. The first component is a pair of *frequency specifications* (f_p, f_f). The *polling frequency* f_p indicates the times at which data source is to be polled in order to detect changes.

The *filter frequency* f_f indicates the times at which new changes should be evaluated and reported to the user. Examples of frequency specifications are "every Friday at 5:00pm" and "every 10 minutes." The polling frequency implies a sequence of time instants $(t_1, t_2, t_3, \ldots)$, which we call *polling times*. *Filter times* are defined analogously. (In the actual system, we also consider two other modes: one in which the polling and/or filter times are obtained following explicit user requests, and the other in which they are obtained as a result of a trigger on the source database firing, if the source provides such a triggering mechanism. To simplify the presentation, we will not describe these modes further here.)

The second component of a subscription is a Lorel query Q_l, which we call the *polling query*. QSS sends the polling (Lorel) query to the wrapper or mediator at the polling times $(t_1, t_2, t_3, \ldots)$ to obtain results $(R_1, R_2, R_3, \ldots)$. An example polling query is the following. (Recall from Section 7.4.1 that "#" is a special character that matches any sequence of zero or more labels in a path. We also use the Lorel operator `like` for string matching.)

```
define polling query LyttonRestaurants as
     select guide.restaurant
     where guide.restaurant.address.# like "%Lytton%";
```

Let R_0 be the empty OEM database, and let R_i be the result of the polling query on the source at time t_i for $i = 1, 2, \ldots$. Each R_i (a Tsimmis query result) is a tree-structured OEM database. Using differencing techniques described in Chapters 4, 5, and 6, QSS obtains a history $H = (t_1, U_1), (t_2, U_2), \ldots$ corresponding to the sequence of OEM databases $(R_0, R_1, R_2, \ldots)$. That is, $U_i(R_{i-1}) = R_i$ for all $i > 0$. Then, QSS constructs a DOEM database $D(R_0, H)$ corresponding to this history H and the initial snapshot R_0, as described in Section 7.3. Thus, intuitively, in the first timestep the results of the polling query are all "created." Thereafter, each subsequent timestep annotates the DOEM database with the changes to the result of the polling query since the previous timestep. We identify the DOEM database corresponding to a polling query using the name of the polling query. Thus the name of the DOEM database corresponding to the above polling query is "`LyttonRestaurants`."

The third component of a subscription is a Chorel query Q_c, called the *filter query*, over the generated DOEM database. In addition to standard Chorel, in Q_c we can use a special time variable "`t[0]`" to refer to the current filter time t_k, and we can use "`t[-1]`," "`t[-2]`," etc., to refer to the past filter times t_{k-1}, t_{k-2}, etc. (If the current filter time is t_k, we define `t[-i]` to be t_{k-i} if $i < k$, and t_I otherwise, where t_I is the initial timestamp associated with the DOEM database of the subscription.) The filter query describes the data and changes of interest to the user. An example filter query is the following:

```
define filter query NewOnLytton as
     select R.name, C.name
     from LyttonRestaurants.restaurant<cre at T1> R
          LyttonRestaurants.cafe<cre at T2> C
     where R.parking = C.parking and T1 > t[-1] and T2 >= 1Jan97;
```

Given our definition of the DOEM database "`LyttonRestaurants`," this query indicates that the user should be notified of the names of restaurant-cafe pairs on Lytton street that share a parking area, where the restaurant was newly created since the last filter time and the cafe was created some time after January 1, 1997. At each filter time t_k ($k > 0$) given by the filter frequency, QSS evaluates Q_c over the DOEM database $D(R_0, H_k)$, where $H_k = (t_1, U_1), \ldots, (t_j, U_j)$, and t_j is the greatest polling time less than t_k, and returns the results to the user.

Example 7.7.1 Consider again the changes to the Guide data described in Example 7.2.2, as depicted in Figure 7.2. Suppose we are interested in being notified every night of new restaurants created in the Guide database since the previous night. We issue the subscription $S = \langle f, Q_l, Q_c \rangle$, where the frequency specification f is "every night at 11:30pm," and the polling query Q_l and filter query Q_c are `Restaurants` and `NewRestaurants` (respectively) as defined below:

```
define polling query Restaurants as
     select guide.restaurant;
```

```
define filter query NewRestaurants as
    select Restaurants.restaurant<cre at T>
    where T > t[-1];
```

Suppose we create this subscription S on December 30th, 1996, at 10:00am. The polling times given by our frequency specification are t_1 = *30Dec96*, t_2 = *31Dec96*, t_3 = *1Jan97*, and so on (all at 11:30pm). At polling time t_1, QSS sends the polling query Q_l to the Guide OEM database, to obtain the result R_1 consisting of the two restaurant objects in Figure 7.1. Since R_0 is the empty OEM database by definition, both restaurant objects will have a *cre* annotation in the DOEM database built by QSS. These annotations all have a timestamp t_1, while the variable `t[-1]` in the query Q_c has value negative infinity at t_1. Therefore, evaluating the filter query Q_c on this DOEM database returns the two restaurant objects as the initial results to the user.

At polling time t_2, the Guide database is unchanged, so the result R_2 of the polling query is identical to R_1. Consequently, no changes are made to the DOEM database maintained by QSS. Note also that at time t_2, $\texttt{t}[-1] = t_1$, so that the create annotations on the restaurant objects in the DOEM database no longer satisfy the predicate `T > t[-1]` in the `where` clause of Q_c. Therefore, the result of Q_c is empty, and the user does not receive any notification.

Before polling time t_3, the Guide database is modified by the addition of a new restaurant object, with name "Hakata," as described in Example 7.2.2. Therefore, at t_3, the result R_3 of the polling query contains the new restaurant object in addition to the two old restaurant objects. The new restaurant object is detected by the differencing algorithm. Accordingly, the DOEM database maintained by QSS now includes the new restaurant object, with a create annotation $cre(t_3)$ on it. Note also that at this time, $\texttt{t}[-1] = t_2$, so that this create annotation satisfies the predicate in the where clause of Q_c. Therefore the result of the query Q_c over the modified DOEM database contains the new restaurant object "Hakata," and the user is notified of this result. □

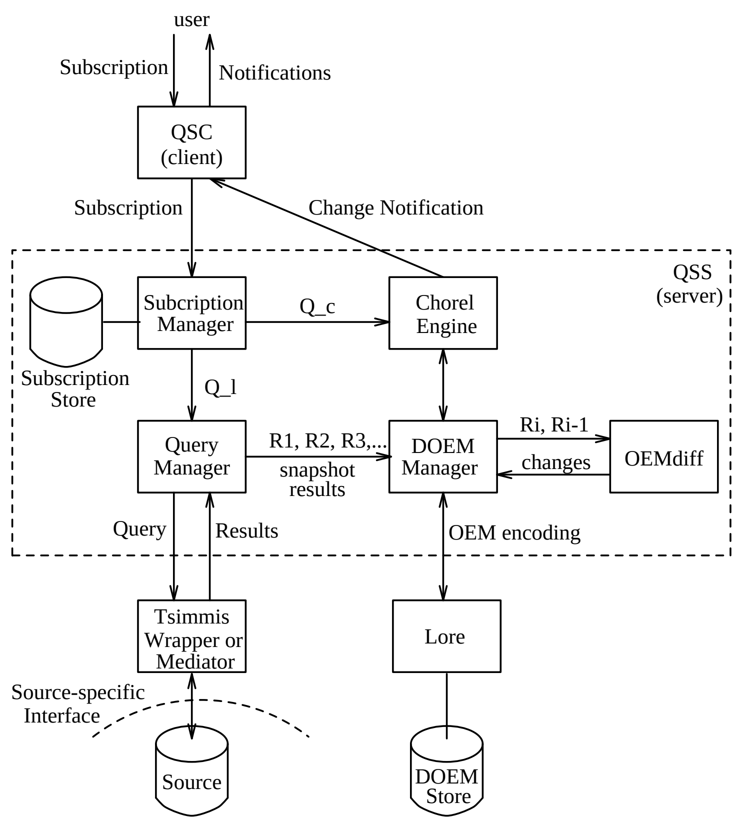


Figure 7.10: System architecture of QSS

7.7.1 QSS Implementation

We now provide a brief discussion of some aspects of our implementation of the Query Subscription Service. Refer to Figure 7.10. The system has a client-server architecture, with one or more client processes (*Query Subscription Clients*, or QSCs) that interact with users, and a server process (QSS) that implements the core functionality. A single server process serves multiple clients. QSC implements a user interface that supports subscription creation and deletion, and also delivers notifications to the user. The QSS server is the principal component of the QSS system. It consists of five main modules:

- The *Subscription Manager* handles all the information relevant to subscriptions. For each subscription, the Subscription Manager maintains the polling query Q_l, the filter query Q_c, the frequency specification f, the identifier of the current DOEM database (stored in the *DOEM Manager* described below), as well as information such as the user name, host name, etc.

- The *Query Manager* module is responsible for sending polling queries to the Tsimmis wrapper or mediator and collecting the resulting OEM results; it interfaces with the Tsimmis CSL library [CGMH+94].

- The *OEMdiff* module implements the differencing algorithm in Chapter 4 to compute the history from the snapshot results of the polling query.

- The *DOEM Manager* maintains the DOEM database corresponding to the sequence of polling query results, using the OEMdiff module to compute changes between successive polling query results. It uses the Lore system [MAG+97] to store OEM encodings of DOEM databases, using the translation scheme described in Section 7.5.1.

- The *Chorel Engine* evaluates the Chorel filter query Q_c for each subscription over the corresponding DOEM database. It includes a preprocessor that replaces the special time variables `t[i]`, if any, in the filter query with the appropriate timestamps as explained above.

The arrows in Figure 7.10 depict the flow of information in QSS. For each subscription, the Subscription Manager uses a timer to invoke the Query Manager with the polling query Q_l at each polling time t_i. The Query Manager communicates with the Tsimmis wrapper or mediator to execute the polling query and to retrieve the result R_i. This result is sent to the DOEM Manager, which forwards R_i to the OEMdiff module along with the previous results R_{i-1}, obtained from the current snapshot of the DOEM database for this subscription. (Alternatively, the DOEM Manager could store the previous result in addition to the DOEM database, thereby trading space for time.) The OEMdiff module compares R_{i-1} with R_i to produce the change operations U such that $U(R_{i-1}) = R_i$. The DOEM Manager then incorporates these changes into the DOEM database for this subscription. Finally, the Chorel filter query Q_c for this subscription is executed over the updated DOEM database by the Chorel Engine, and the results are sent to the user via the QSC client.

For certain polling queries, QSS may need to store a large portion of the underlying database in order to detect changes accurately. We are exploring the following ways of limiting the space used for storing DOEM databases: (1) merging the DOEM databases for several subscriptions that have similar polling queries; (2) making the client responsible for storing the DOEM databases for its subscriptions; and (3) trading accuracy for space by storing a smaller state at the expense of not being able to detect all changes accurately. We are also working on methods for determining a polling query and filter query automatically from a simpler form of subscription query.

7.8 Summary

In this chapter we studied the problem of representing, storing, and querying historical data in the context of heterogeneous, autonomous databases. We motivated the need for a uniform representation scheme for changes in semistructured data, and for a query language that allows direct access to changes. We presented a simple data model, DOEM, for representing historical semistructured data. In DOEM, changes to data items are represented using annotations on the affected data, making DOEM particularly well-suited to browsing historical semistructured data marked up with changes.

In addition to browsing, the DOEM data model also supports a historical query language called Chorel. An important feature of Chorel is that changes are treated as first-class, allowing data and changes to be queried in an integrated manner. Our implementation of a database system for historical semistructured data, called CORE, is based on the Lore database system for semistructured data. We described how we implemented CORE as an extension to Lore by using a technique that encodes DOEM in OEM and translates Chorel queries on a DOEM database to Lorel queries on its DOEM encoding. Apart from modularity, this implementation strategy makes our techniques easily adaptable to other database systems for structured and semi-structured data.

While data items in the DOEM model have four basic kinds of annotations describing their history of changes, we can also associate additional virtual annotations with data items. Such virtual annotations are similar to views in traditional database systems, and can be used for the analogous purpose of providing convenient, customized access to data. We demonstrated how we use virtual annotations to facilitate snapshot-based access to a historical database. We defined the semantics of snapshot-based virtual annotations such as `<snap>` in terms of the base annotations, and described extensions to our translation-based implementation scheme to accomodate queries containing such annotations.

We also described the design and implementation of a Query Subscription Service (QSS) that allows us to subscribe to interesting changes in source databases. To specify interesting changes, QSS uses a general and powerful subscription language based on Chorel. We defined the syntax and semantics of this language, and described its implementation based on our implementations of OEMDiff and CORE. Together with the techniques for detecting changes described in Chapters 4, 5, and 6, the techniques of this chapter are the basis of our implementation of the C^3 change management system described in the next chapter.

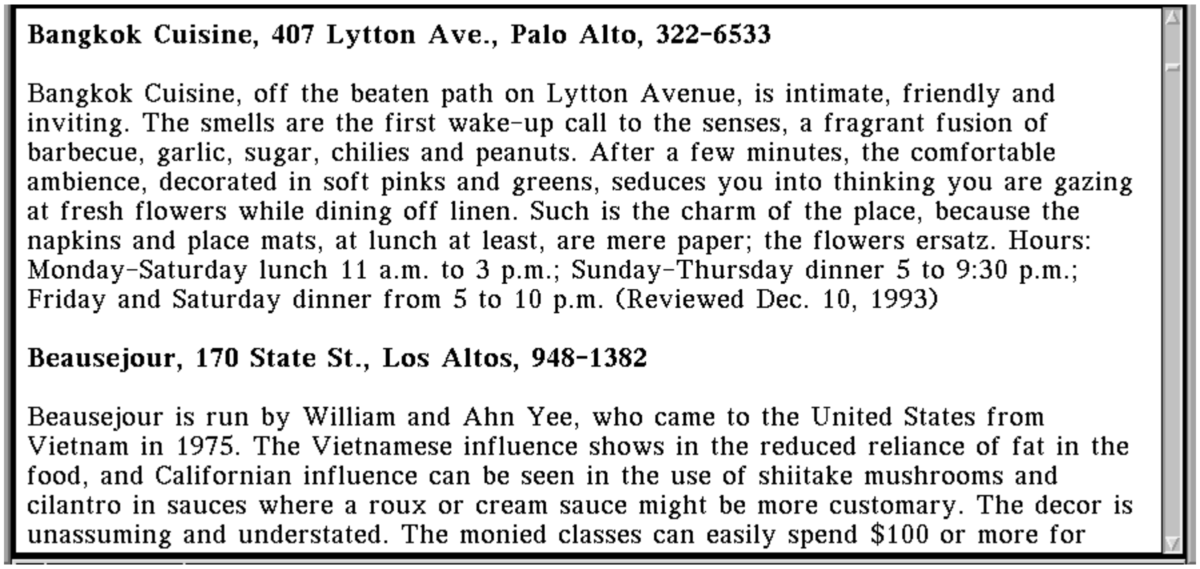
Chapter 8

System Implementation

In this chapter, we describe how we have used the techniques described in previous chapters to implement the C^3 system for managing change in heterogeneous, autonomous databases. In Section 8.1 we describe the functionality provided by the C^3 system to its users. Using an extended example, we illustrate how the C^3 system may be used. In Section 8.2, we describe how this functionality is implemented. Continuing with the extended example, we present the system response to a representative set of user and external events. Recall that we outlined the high-level architecture of our system in Chapter 3. Further, in Chapters 4–7 we described the design and implementation of the major system modules. Therefore, in this chapter, we discuss only those system implementation details that are not presented in earlier chapters. In particular, we focus on describing how system modules interact with each other to provide the overall system functionality.

8.1 User Interactions

A quick glance at Figure 8.15 (described in detail in Section 8.2) suggests that there are three major user interfaces to the C^3 system, one each for the three principal modules: TDiff, CORE, and QSS. Although the QSS interface is the most comprehensive of the three, using the TDiff and CORE interfaces separately is often useful. For example, while QSS is restricted to presenting all data in our integrating data



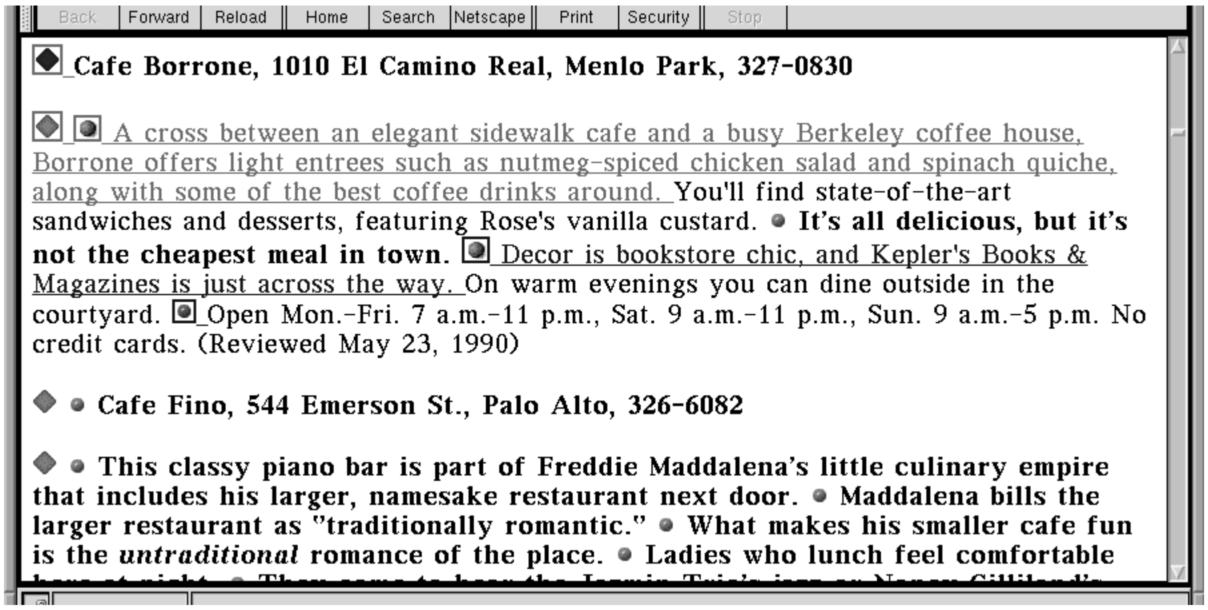
Bangkok Cuisine, 407 Lytton Ave., Palo Alto, 322-6533

Bangkok Cuisine, off the beaten path on Lytton Avenue, is intimate, friendly and inviting. The smells are the first wake-up call to the senses, a fragrant fusion of barbecue, garlic, sugar, chilies and peanuts. After a few minutes, the comfortable ambience, decorated in soft pinks and greens, seduces you into thinking you are gazing at fresh flowers while dining off linen. Such is the charm of the place, because the napkins and place mats, at lunch at least, are mere paper; the flowers ersatz. Hours: Monday-Saturday lunch 11 a.m. to 3 p.m.; Sunday-Thursday dinner 5 to 9:30 p.m.; Friday and Saturday dinner from 5 to 10 p.m. (Reviewed Dec. 10, 1993)

Beausejour, 170 State St., Los Altos, 948-1382

Beausejour is run by William and Ahn Yee, who came to the United States from Vietnam in 1975. The Vietnamese influence shows in the reduced reliance of fat in the food, and Californian influence can be seen in the use of shiitake mushrooms and cilantro in sauces where a roux or cream sauce might be more customary. The decor is unassuming and understated. The monied classes can easily spend $100 or more for

Figure 8.1: Restaurant reviews from the Palo Alto Weekly



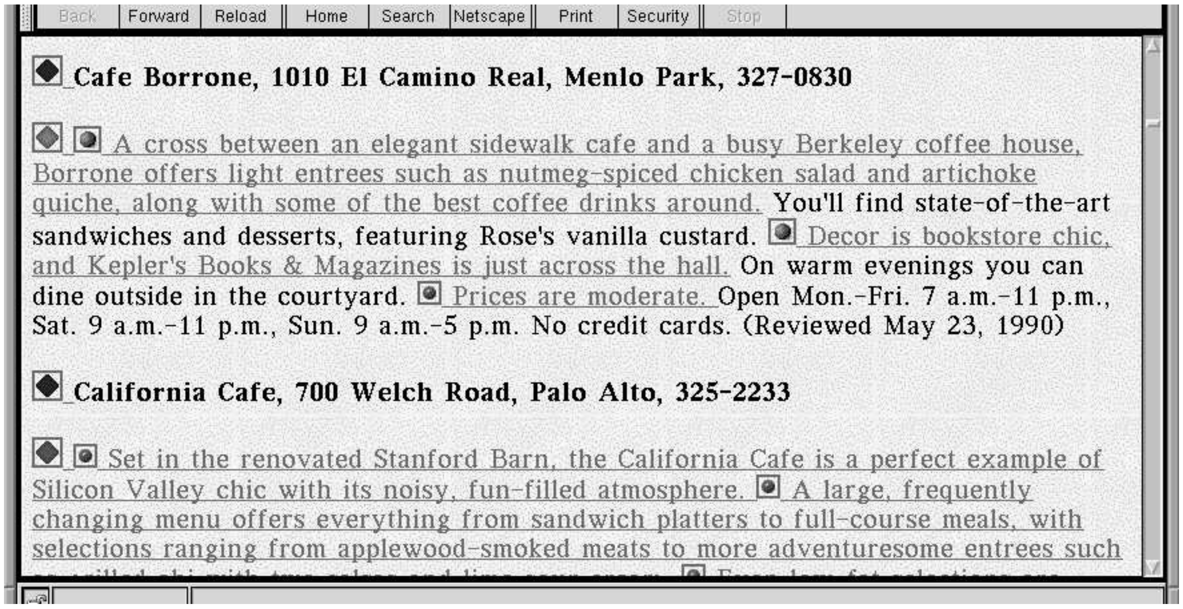
Cafe Borrone, 1010 El Camino Real, Menlo Park, 327-0830

A cross between an elegant sidewalk cafe and a busy Berkeley coffee house, Borrone offers light entrees such as nutmeg-spiced chicken salad and spinach quiche, along with some of the best coffee drinks around. You'll find state-of-the-art sandwiches and desserts, featuring Rose's vanilla custard. ● **It's all delicious, but it's not the cheapest meal in town.** Decor is bookstore chic, and Kepler's Books & Magazines is just across the way. On warm evenings you can dine outside in the courtyard. Open Mon.-Fri. 7 a.m.-11 p.m., Sat. 9 a.m.-11 p.m., Sun. 9 a.m.-5 p.m. No credit cards. (Reviewed May 23, 1990)

● **Cafe Fino, 544 Emerson St., Palo Alto, 326-6082**

● **This classy piano bar is part of Freddie Maddalena's little culinary empire that includes his larger, namesake restaurant next door.** ● **Maddalena bills the larger restaurant as "traditionally romantic."** ● **What makes his smaller cafe fun is the *untraditional* romance of the place.** ● **Ladies who lunch feel comfortable**

Figure 8.2: New version of reviews with changes marked



Cafe Borrone, 1010 El Camino Real, Menlo Park, 327-0830

A cross between an elegant sidewalk cafe and a busy Berkeley coffee house, Borrone offers light entrees such as nutmeg-spiced chicken salad and artichoke quiche, along with some of the best coffee drinks around. You'll find state-of-the-art sandwiches and desserts, featuring Rose's vanilla custard. Decor is bookstore chic, and Kepler's Books & Magazines is just across the hall. On warm evenings you can dine outside in the courtyard. Prices are moderate. Open Mon.-Fri. 7 a.m.-11 p.m., Sat. 9 a.m.-11 p.m., Sun. 9 a.m.-5 p.m. No credit cards. (Reviewed May 23, 1990)

California Cafe, 700 Welch Road, Palo Alto, 325-2233

Set in the renovated Stanford Barn, the California Cafe is a perfect example of Silicon Valley chic with its noisy, fun-filled atmosphere. A large, frequently changing menu offers everything from sandwich platters to full-course meals, with selections ranging from applewood-smoked meats to more adventuresome entrees such

Figure 8.3: Old version of reviews with changes marked

model (OEM), TDiff is free to present data in its native format such as plain text, HTML, and Latex.

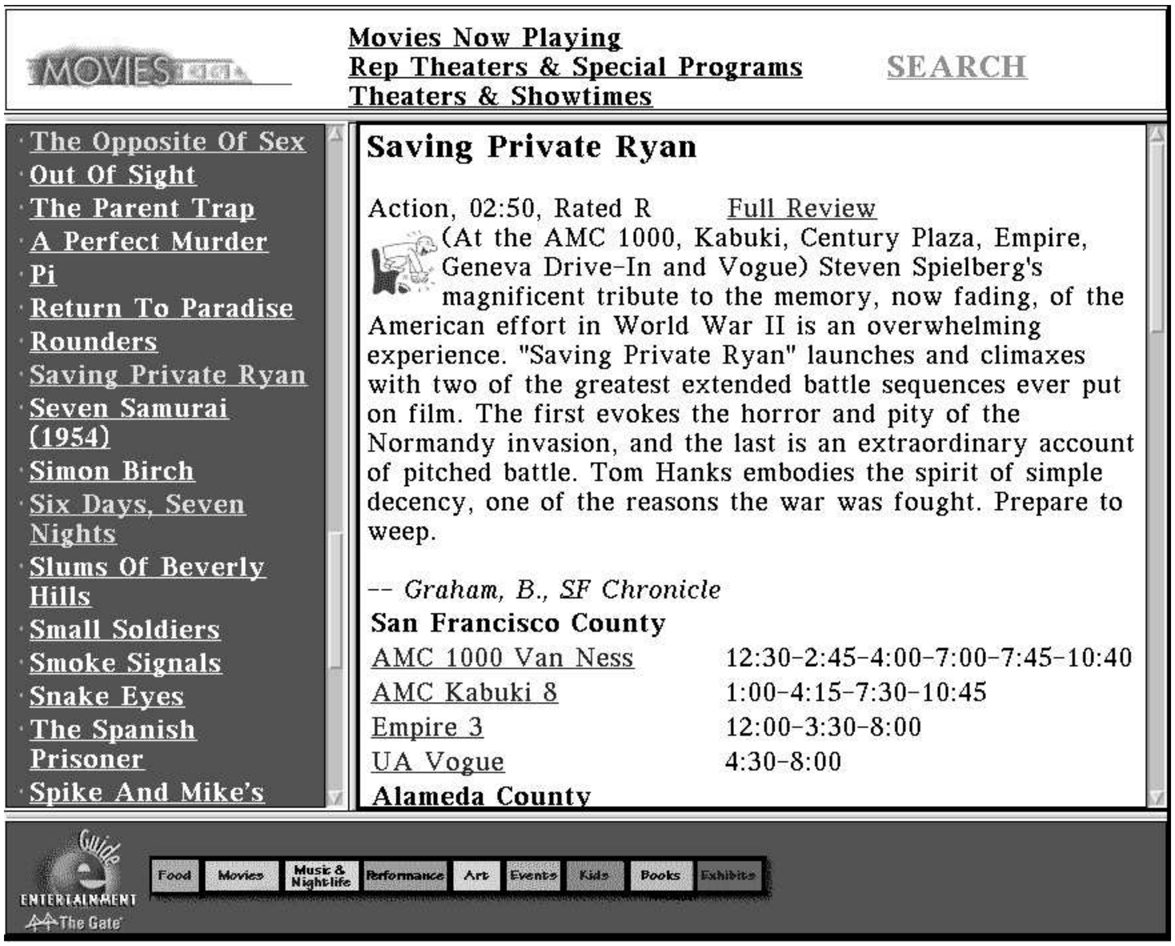
8.1.1 Using TDiff

When presented with two snapshots of data, such as two versions of a Web page, TDiff computes the differences between these snapshots, and presents the results in a graphical format that can be conveniently browsed. For example, suppose we are interested in finding out what has changed in a Web page that lists approximately 200 restaurant reviews from the Palo Alto Weekly [PAW98]. Figure 8.1 shows an excerpt from this page. Suppose we are interested in comparing the version of this Web page from January 1994 with that from November 1995. A version of TDiff specialized for HTML data takes these Web pages as input, parses them into ordered trees, and computes the differences between them using the methods of Chapter 4. The insertions, deletions, updates, and moves thus detected are then displayed using icons of different colors. TDiff marks changes in both the old and new versions of the document using representative icons. Figures 8.2 and 8.3 depict excerpts from the marked-up new and old versions of the restaurant reviews. In the interactive system, the old version is presented with a yellow background in order to clearly distinguish it from the new one. Similarly, the icons use different colors to represent insertions, deletions, updates, and moves. By clicking on one of these icons, one can find out more about the change it represents. When one clicks on an icon in one version, the corresponding information from the other version is displayed. For example, clicking on a red dot, which denotes a deleted sentence, results in the display of the old version of the document with the deleted sentence highlighted.

We have implemented a few special versions of TDiff for data formats such as plain text, simple HTML, and Latex. In general, the method for studying differences between versions of data is to first convert the data to OEM format, and then use the OEM version of TDiff. The TDiff interface allows us to compare only two versions of data at a time. For a more general solution, we use the QSS interface described next.

8.1.2 Using QSS

Recall, from Chapter 7, that the *Query Subscription Service (QSS)* module acts as a driver for the C^3 system and provides a flexible and general method to monitor changes to autonomous databases using a subscription metaphor. We have interfaced our C^3 system with a number of source databases, including a Web site with traffic reports, a Web site listing entertainment events, and a bibliographic server. The user first connects to the QSS server for the source database of interest. For our example, we use the *eGuide* Web site, which lists entertainment events for the San Francisco Bay Area. Figure 8.4 shows a screen-shot from this site. The figure suggests the natural hierarchical structure of this Web database, consisting of categories such as movies, restaurants, and events (depicted near the bottom of the figure), subcategories such as Movies Now Playing, Special Programs, and Showtimes (depicted near the top), individual listings such as the movie listings (depicted on the left), details of the movie (depicted on the right), and so on. Each movie listing contains information such as the rating, running time, critics rating, MPAA rating, a review, and a listing of



Saving Private Ryan

Action, 02:50, Rated R

(At the AMC 1000, Kabuki, Century Plaza, Empire, Geneva Drive-In and Vogue) Steven Spielberg's magnificent tribute to the memory, now fading, of the American effort in World War II is an overwhelming experience. "Saving Private Ryan" launches and climaxes with two of the greatest extended battle sequences ever put on film. The first evokes the horror and pity of the Normandy invasion, and the last is an extraordinary account of pitched battle. Tom Hanks embodies the spirit of simple decency, one of the reasons the war was fought. Prepare to weep.

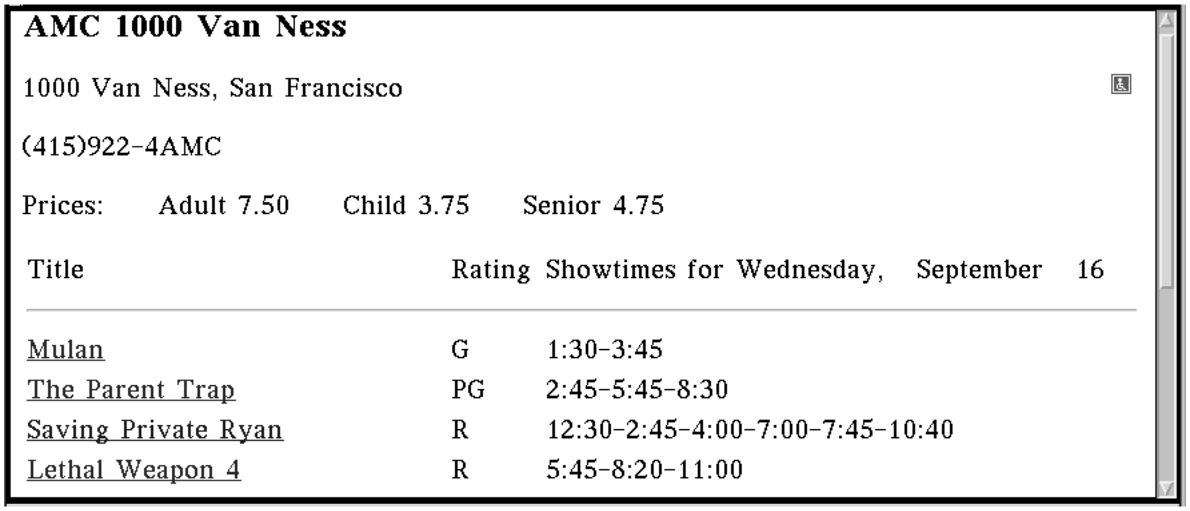
-- *Graham, B., SF Chronicle*

San Francisco County

AMC 1000 Van Ness	12:30-2:45-4:00-7:00-7:45-10:40
AMC Kabuki 8	1:00-4:15-7:30-10:45
Empire 3	12:00-3:30-8:00
UA Vogue	4:30-8:00

Alameda County

Figure 8.4: The eGuide Web database: movie section



AMC 1000 Van Ness

1000 Van Ness, San Francisco

(415)922-4AMC

Prices: Adult 7.50 Child 3.75 Senior 4.75

Title	Rating	Showtimes for Wednesday, September 16
Mulan	G	1:30-3:45
The Parent Trap	PG	2:45-5:45-8:30
Saving Private Ryan	R	12:30-2:45-4:00-7:00-7:45-10:40
Lethal Weapon 4	R	5:45-8:20-11:00

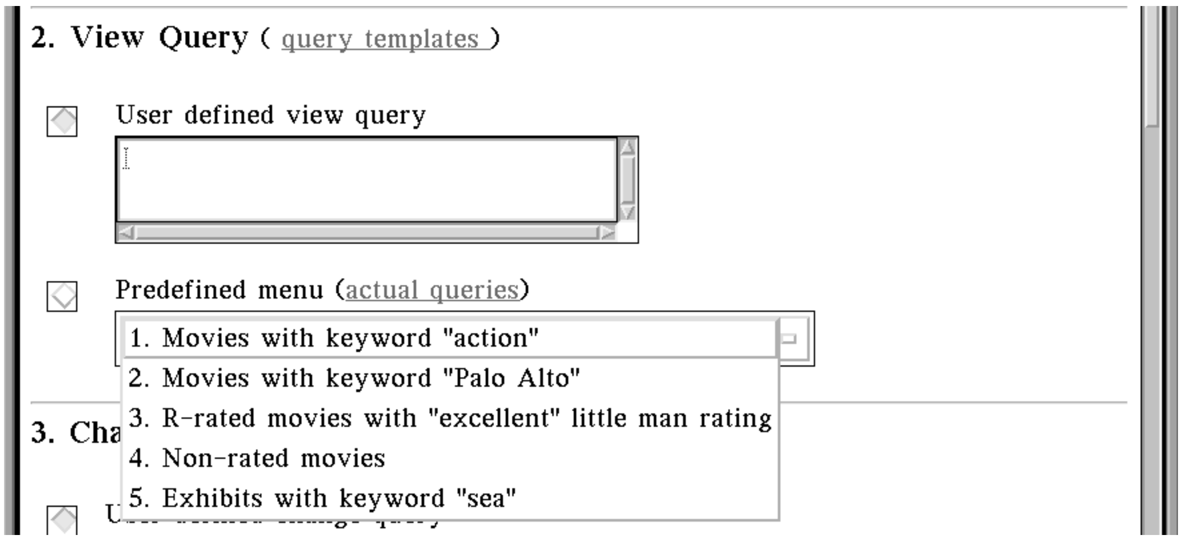
Figure 8.5: The eGuide Web database: theater details

theaters that are screening the movie. Clicking on some of these items results in more detailed information. For example, Figure 8.5 depicts an excerpt of the information displayed when one clicks on a theater name.

After logging into the QSS server for eGuide, the user is presented with the option of reviewing her present subscriptions or creating a new one. We first describe the process of creating a subscription. Elaborating on the formal definition of a subscription in Chapter 7, a subscription consists of a unique name, and the following components:

Polling Query: Intuitively, this query describes the portion of the source database that is of interest to the user. More precisely, the polling query is a Lorel query sent periodically by QSS to the wrapper of the source database in order to detect changes and generate a history. Each wrapper supports a limited set of query types, and these are encoded using a list of *query templates* presented to the user. Commonly used instantiations of these templates are also presented using a menu.

Figure 8.6 shows the five commonly used polling queries offered for eGuide.



2. View Query (query templates)

User defined view query

Predefined menu (actual queries)

1. Movies with keyword "action"
2. Movies with keyword "Palo Alto"
3. R-rated movies with "excellent" little man rating
4. Non-rated movies
5. Exhibits with keyword "sea"

3. Cha

Figure 8.6: Menu of common polling queries for eGuide

```
1. Get all movies whose titles, descriptions, etc. contain the keyword $N.
   select movie where keyword(movie, $N);

2. Get all movies with title $N.
   select movie where movie.title = $N;

3. Get all movies whose titles contain the word $N.
   select movie where contains(movie.title,  $N);

4. Get all movies with rating $N.
      Rating values can be "G", "PG", "PG-13", "R", "NC-17" or "NR".
   select movie where movie.rating = $N;

5. Get all movies whose ratings are higher than or equal to $N.
   select movie where rating_ge(movie, $N);
```

Figure 8.7: Some polling query templates from the eGuide wrapper

QSS also offers a list of twelve query templates for movies and four templates for special events. An excerpt from this list of query templates is displayed in Figure 8.7. The listing includes the English query, and its equivalent Lorel version. (The templates also include MSL versions of each query; these are not shown in the figure.) For our example, we select the "non-rated movies" option from the menu.

Filter Query: Intuitively, this query describes the changes of which the user wishes to be notified. More precisely, the filter query is a Chorel query that is periodically evaluated over the DOEM database representing the historical data generated by polling queries. This DOEM database is given the reserved name `ViewRoot`. As described in Chapter 7, the filter query can also make use of the special syntax `t[-i]` to refer to past query evaluation times. Similarly, past polling query evaluation times are accessed using the syntax `t'[-i]`. Commonly used filter queries are presented in a menu. Note that since every Lorel query is also a Chorel query, the Chorel query is not required to refer to changes, although it is typically more useful when it does.

Figure 8.8 lists the five commonly used filter queries offered in the menu for eGuide. For our example, we write the following Chorel query which returns the titles of newly added movies along with the times they were added:

```
select X,T from ViewRoot.<add at T>%.title X where T > t[-1];
```

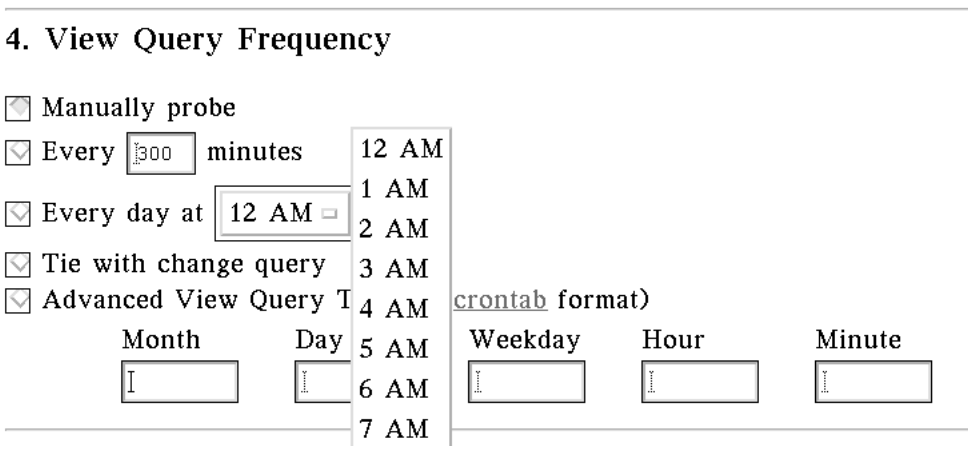
Polling Frequency: This frequency specifies when the source database is to be polled for new changes. The user can select from a menu of commonly used frequencies as indicated by the screen-shot in Figure 8.9. More generally, we use the syntax of the Unix *cron* utility to specify the frequency [Vix98]. A special value for the polling frequency is `Probe`, which indicates that polling is to be performed on explicit user request only.

For our example, we select from the menu "every day at midnight."

Filter Frequency: This frequency is analogous to the polling frequency, and indicates when the filter query is to be evaluated over the DOEM database of the

```
1.Objects added to the top level since the last checking
     select ViewRoot.<add at t>%
     where t > t[-1];
2.Objects newly found in the top level since the last checking
     select ViewRoot.%<cre at t>
     where t > t[-1];
3.Objects removed from the top level since the last checking
     select ViewRoot.<rem at t>%
     where t > t[-1];
4.Objects newly removed from the top level since the last checking
     select X
     from ViewRoot.<rem at t1>%<del> X
     where t1 > t[-1] and
           forall Y in ViewRoot.<add at t2>% :
                          X <> Y or t2 <= t[-1] or t1 < t2;
5.Objects in the top level updated since the last checking
     select ViewRoot.%<upd at t>
     where t > t[-1];
```

Figure 8.8: Menu of common filter queries

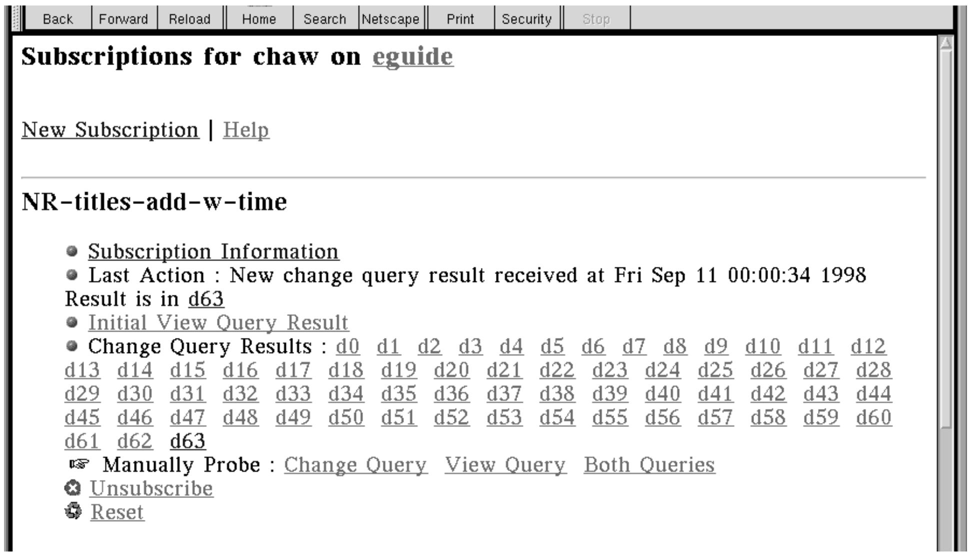


4. View Query Frequency

[✓] Manually probe
[ ] Every 300 minutes
[ ] Every day at 12 AM
[ ] Tie with change query
[ ] Advanced View Query T crontab format)

Month	Day	Weekday	Hour	Minute

Figure 8.9: Specifying the polling frequency



Subscriptions for chaw on eguide

New Subscription | Help

NR-titles-add-w-time

- Subscription Information
- Last Action : New change query result received at Fri Sep 11 00:00:34 1998 Result is in d63
- Initial View Query Result
- Change Query Results : d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12 d13 d14 d15 d16 d17 d18 d19 d20 d21 d22 d23 d24 d25 d26 d27 d28 d29 d30 d31 d32 d33 d34 d35 d36 d37 d38 d39 d40 d41 d42 d43 d44 d45 d46 d47 d48 d49 d50 d51 d52 d53 d54 d55 d56 d57 d58 d59 d60 d61 d62 d63
- Manually Probe : Change Query View Query Both Queries
- Unsubscribe
- Reset

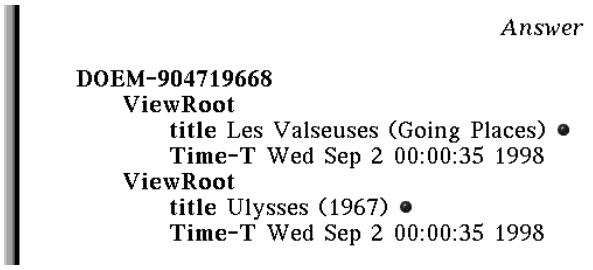
Figure 8.10: QSS subscription review screen

subscription in order to detect new changes of interest. Similar to the polling frequency, the filter frequency may have the special value `Probe`, indicating that the filter query is to be evaluated on explicit user request only. In addition, the filter query may have the special value `Tie`, indicating that the filter query is to be evaluated immediately after each evaluation of the polling query.

For our example, we select the Tie option from the menu.

When we request the creation of a subscription named *NR-titles* with the parameters selected above, QSS evaluates the polling and filter query once immediately in order to establish a baseline for future changes. The results of the filter query evaluation are returned as the initial result of the subscription. Every midnight following the creation of this subscription, QSS executes the polling query, updates the DOEM database based on its results, and checks for new changes satisfying the filter query. New results are stored for review by the subscription owner. (Optionally, the subscription owner may request an email notification.)

Figure 8.10 shows an excerpt from the *subscription review screen*, focusing on the NR-titles subscription created above. As shown in the figure, QSS presents the set of



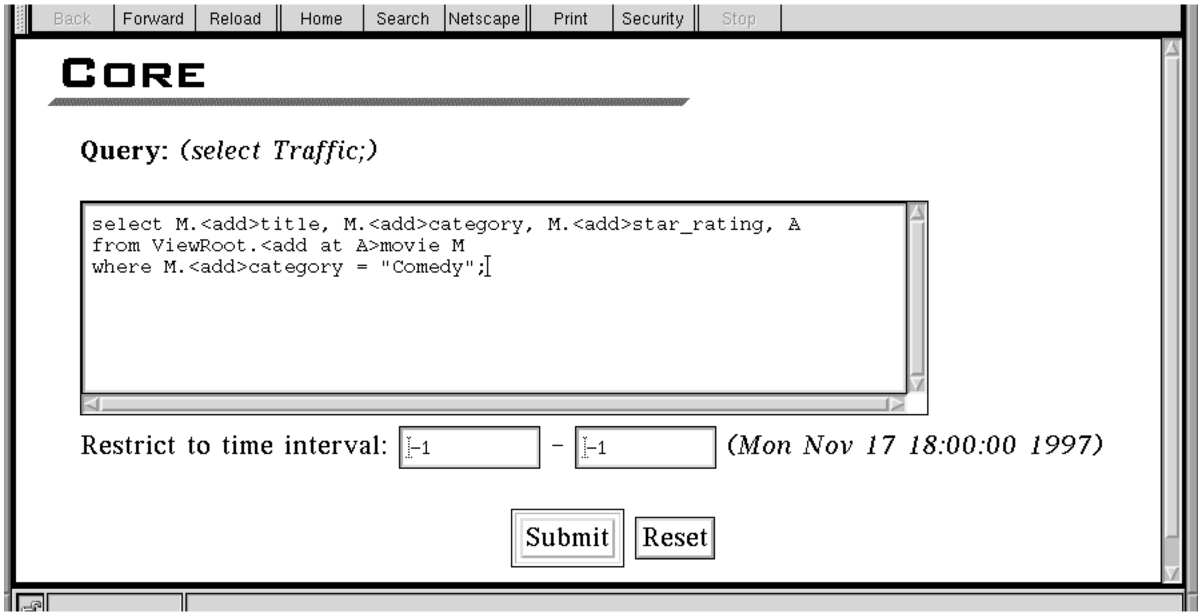
```
                                   Answer

DOEM-904719668
    ViewRoot
        title Les Valseuses (Going Places) ●
        Time-T Wed Sep 2 00:00:35 1998
    ViewRoot
        title Ulysses (1967) ●
        Time-T Wed Sep 2 00:00:35 1998
```

Figure 8.11: A result for the subscription NR-titles

accumulated filter (change) query results. By clicking on any of the result identifiers, we can view the detailed results. Figure 8.11 is an excerpt from the *results screen* displaying the result delivered for NR-titles on the 2nd of September. Recall that our filter query asks for the titles of newly added NR movies along with the times at which their additions were detected. The figure shows the titles of two newly added movies and the corresponding times. The results screen displayed by QSS is an active one, meaning we can click on links and icons to browse the result in more detail by navigating the DOEM database of this subscription. As discussed below, we can also evaluate arbitrary Chorel queries on this database.

8.1.3 Using CORE

Recall, from Chapter 7, that the *Change Object Repository (CORE)* module is our implementation of a historical database system for OEM data. Figure 8.12 displays a screen-shot of the CORE **query screen**. In addition to typing in a Chorel query, we can also restrict the browsable results to some time interval. The semantics of such *timestamp-restricted browsing* are as follows: Browsing a DOEM database D with a timestamp-restriction $(B, E]$ disregards all nodes and arcs in D that are not present in at least one OEM snapshot $O_t(D)$ for some timestamp $t \in (B, E]$. (Recall that we defined the OEM snapshot of a DOEM database in Section 7.3.2 of Chapter 7.) Note that the timestamp restriction, if any, on browsing of the results of a query does not change the semantics of query evaluation. That is, the given Chorel query is



Query: *(select Traffic;)*

```
select M.<add>title, M.<add>category, M.<add>star_rating, A
from ViewRoot.<add at A>movie M
where M.<add>category = "Comedy";
```

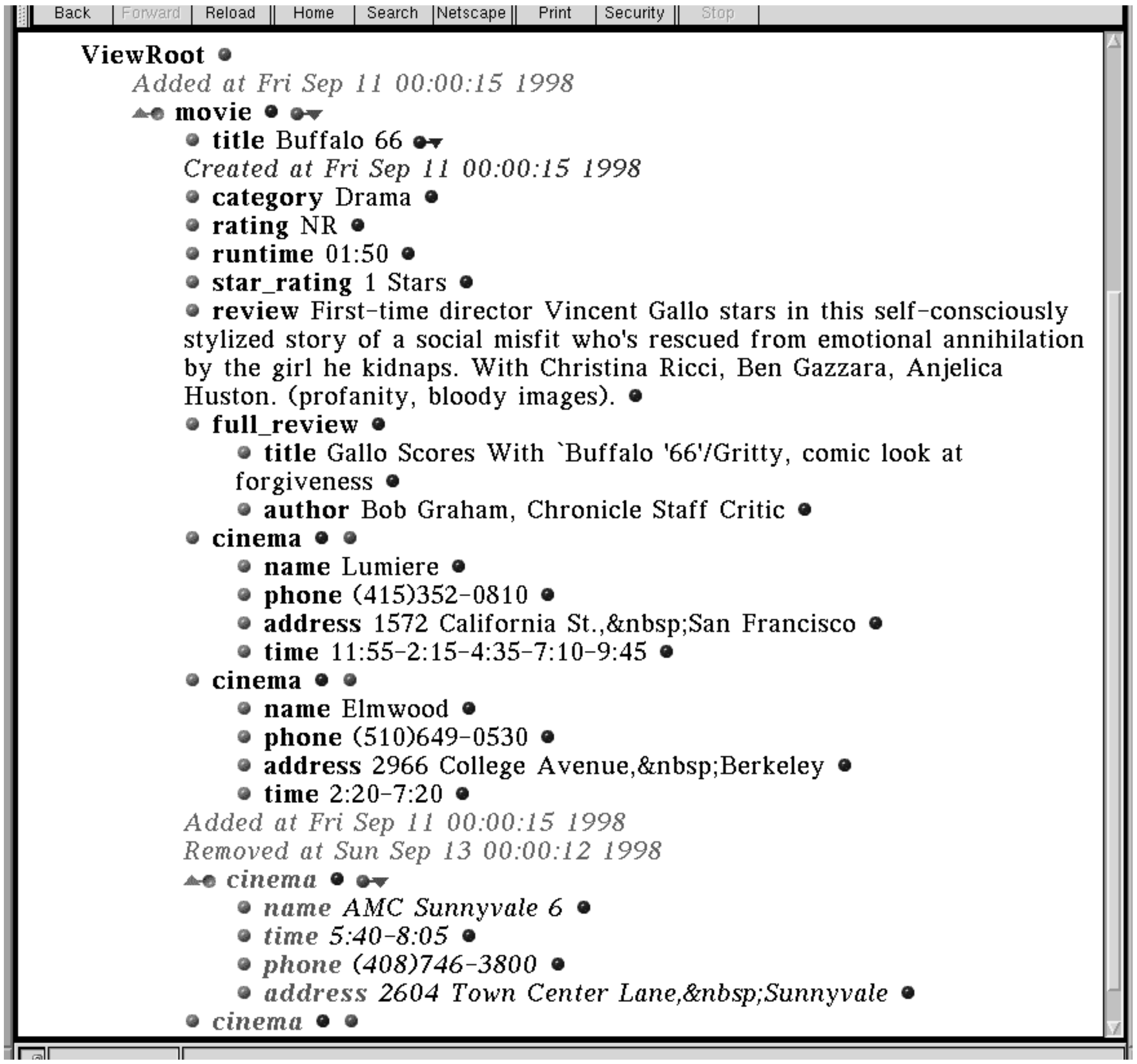
Restrict to time interval: -1 – -1 *(Mon Nov 17 18:00:00 1997)*

Figure 8.12: CORE query interface

evaluated as before to yield a set of object identifiers (OIDs) as result. It is only the subsequent browsing of the DOEM database with these OIDs as starting points that is modified as described above. In addition to being a useful feature for studying past states of a DOEM database, timestamp-restricted browsing is essential to our method for storing, retrieving, and browsing QSS results, as described in the Section 8.2.

Using the CORE query interface, we can interactively browse and query any DOEM database. In particular, we can browse and query the DOEM database implicitly created by each QSS subscription, based on the definitions in Section 7.7 of Chapter 7. In brief, this DOEM database encodes the history of polling query results for this subscription. (We discuss the construction of this database in more detail in Section 8.2.) In the context of our subscription NR-titles introduced above, we can write the Chorel query "`select ViewRoot;`" to return the special *named object* that is the root of the DOEM database for this subscription.

Figure 8.13 shows a portion of the resulting **browsing screen**. Since we did not restrict the browsing to a specified time interval, all objects in the DOEM database

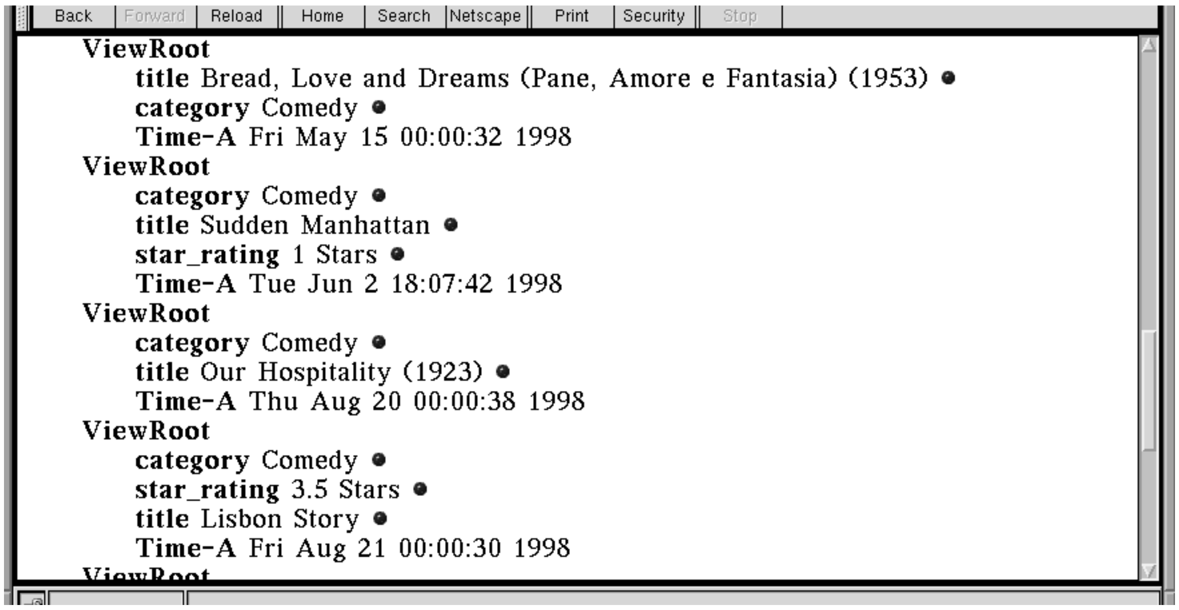


```
ViewRoot ●
    Added at Fri Sep 11 00:00:15 1998
    movie ●
        ● title Buffalo 66
        Created at Fri Sep 11 00:00:15 1998
        ● category Drama ●
        ● rating NR ●
        ● runtime 01:50 ●
        ● star_rating 1 Stars ●
        ● review First-time director Vincent Gallo stars in this self-consciously
        stylized story of a social misfit who's rescued from emotional annihilation
        by the girl he kidnaps. With Christina Ricci, Ben Gazzara, Anjelica
        Huston. (profanity, bloody images). ●
        ● full_review ●
            ● title Gallo Scores With `Buffalo '66'/Gritty, comic look at
            forgiveness ●
            ● author Bob Graham, Chronicle Staff Critic ●
        ● cinema ● ●
            ● name Lumiere ●
            ● phone (415)352-0810 ●
            ● address 1572 California St.,&nbsp;San Francisco ●
            ● time 11:55-2:15-4:35-7:10-9:45 ●
        ● cinema ● ●
            ● name Elmwood ●
            ● phone (510)649-0530 ●
            ● address 2966 College Avenue,&nbsp;Berkeley ●
            ● time 2:20-7:20 ●
        Added at Fri Sep 11 00:00:15 1998
        Removed at Sun Sep 13 00:00:12 1998
        cinema ●
            ● name AMC Sunnyvale 6 ●
            ● time 5:40-8:05 ●
            ● phone (408)746-3800 ●
            ● address 2604 Town Center Lane,&nbsp;Sunnyvale ●
        ● cinema ● ●
```

Figure 8.13: Result of the query "`select ViewRoot;`" on the NR-titles database

are available for browsing by navigating down from the root. The browsing screen displays one object per line (with line wrapping for details that do not fit on a line). The *distinguished objects* that are part of the query result are displayed with the least indentation; other objects reachable from the distinguished objects are displayed with indentation proportional to their distance from the distinguished objects. By default, the display is restricted to objects and links that exist in the *current snapshot* of the DOEM database. (See Section 7.3.2 in Chapter 7 for a discussion of snapshots.) Further, to keep the display manageable, by default only objects reachable from the distinguished objects using a path of length three or less are displayed. Such defaults can be easily modified. On each line representing an object, the last label in the path used to reach the object is displayed in bold font, followed by the value of the object in normal font if it is an atomic object. (Recall, from Chapter 7, that in our data model only atomic objects, which are objects that have no outgoing arcs, have values.) We use a green icon at the beginning of an object's display line to represent the set of arc annotations on the last arc (whose label follows) in the path used to reach the object. Clicking on this icon toggles the detailed display of these arc annotations. Similarly, we use a blue icon at the end of an object's display line to represent the set of arc annotations on that object, and clicking this icon toggles their detailed display. The icons denoting expanded annotation sets have a triangular arrow pointing to the details.

Continuing with our example, the result of our trivial query (`select ViewRoot;`) is the singleton set containing the special named object. The first line in the results display depicted in Figure 8.13 corresponds to this special object. The figure indicates that this ViewRoot object has only one subobject, with label movie. This movie object in turn has a number of self-explanatory subobjects with labels such as title, rating, and cinema. Figure 8.13 also indicates that we have expanded the node annotation for the movie title node, and the arc annotation for the movie arc, and displays the timestamps of these annotations. Red icons next to a node are used to represent the set of outgoing arcs that do not exist in the current snapshot of the DOEM database. Clicking on these red icons toggles the display of such historical arcs, which are hidden by default. In the display shown in Figure 8.13, we have



```
ViewRoot
    title Bread, Love and Dreams (Pane, Amore e Fantasia) (1953) ●
    category Comedy ●
    Time-A Fri May 15 00:00:32 1998
ViewRoot
    category Comedy ●
    title Sudden Manhattan ●
    star_rating 1 Stars ●
    Time-A Tue Jun 2 18:07:42 1998
ViewRoot
    category Comedy ●
    title Our Hospitality (1923) ●
    Time-A Thu Aug 20 00:00:38 1998
ViewRoot
    category Comedy ●
    star_rating 3.5 Stars ●
    title Lisbon Story ●
    Time-A Fri Aug 21 00:00:30 1998
```

Figure 8.14: Result of the query in Figure 8.12 on the NR-titles database

expanded the removed subobjects of the movie object. The last two cinema objects in the figure (shown in red on screen) are displayed as a result of this expansion. To confirm that these cinema objects are indeed removed subobjects of the movie object, we have also expanded the arc annotations on the arc leading to the penultimate cinema object. The figure indicates that this arc was added on September 11th and removed on the 13th. The figure also indicates that we have expanded the removed subobjects of this cinema object. The resulting name, time, phone, and address subobjects are also displayed in red.

As an example of a more interesting query, consider the following, which looks for movies in the Comedy category and returns their titles, categories, star rating (critic's rating), and time of addition to the database:

```
select M.<add>title, M.<add>category, M.<add>star_rating, A
from ViewRoot.<add at A>movie M
where M.<add>category = "Comedy";
```

Figure 8.14 depicts a portion of the result of this query on the DOEM database of

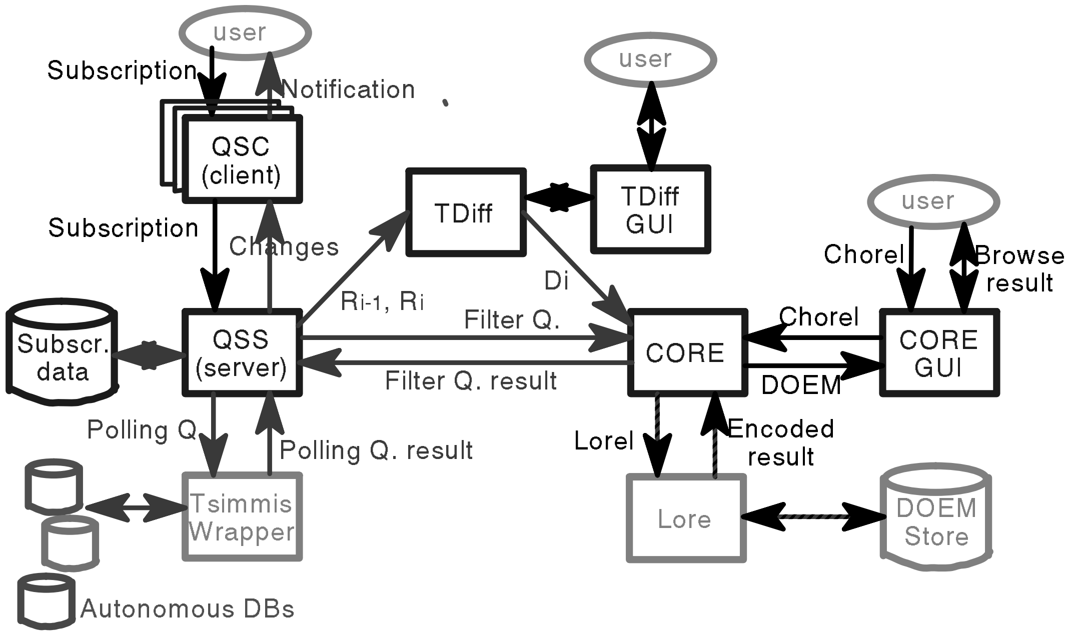
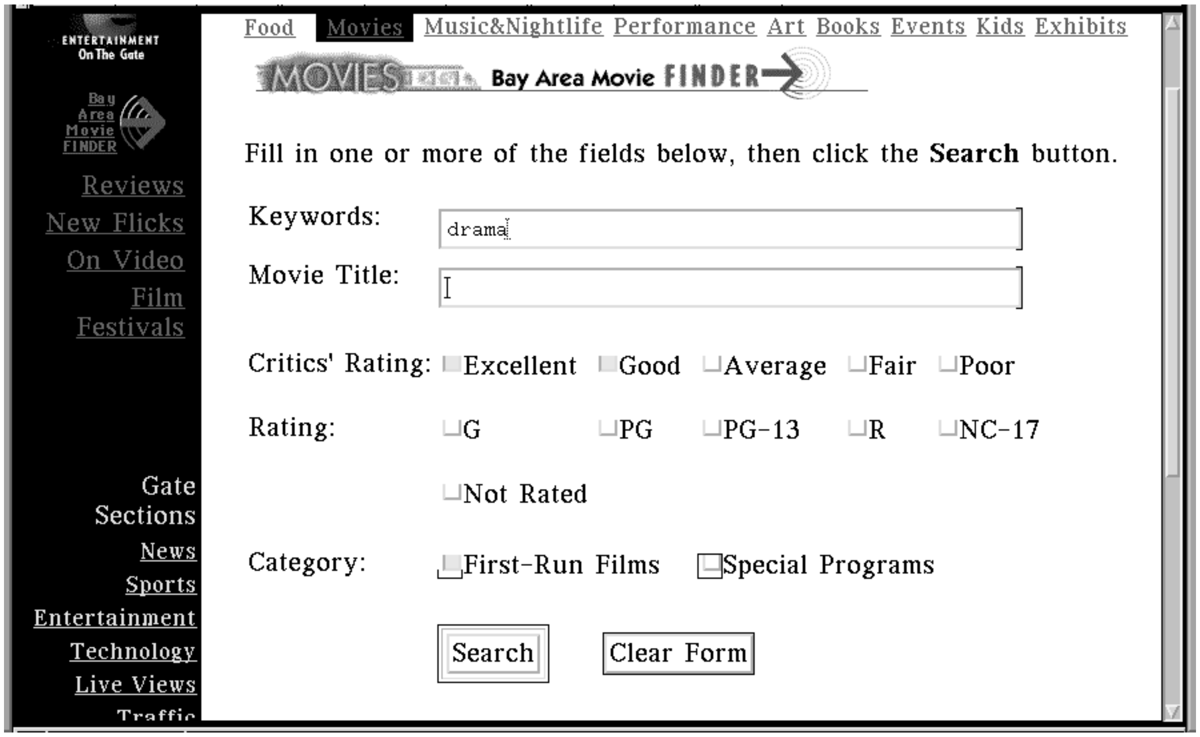


Figure 8.15: Architecture of the C^3system

our NR-titles subscription. In keeping with the semistructured nature of the data we model, it is likely that one or more kinds of subobjects may be missing from some movie objects in the database. However, our use of a semistructured query language allows us to conveniently interact with the database in spite of such missing data. For example, observe that some of the listings in the result do not have star-ratings; these listings correspond to movie objects in the database that have missing star-rating subobjects. As with our earlier example, we can interactively browse these results by expanding annotations, exposing removed subobjects, and so on. CORE thus provides a powerful and convenient method for iteratively querying and browsing a historical semistructured database.

8.2 System Interactions

In the previous section, we described the functionality offered by the C^3 system through the TDiff, QSS, and CORE interfaces. In this section, we describe how this



Fill in one or more of the fields below, then click the **Search** button.

Keywords:	drama
Movie Title:	
Critics' Rating:	[✓ Excellent] [✓ Good] Average Fair Poor
Rating:	G PG PG-13 R NC-17 Not Rated
Category:	[✓ First-Run Films] Special Programs

Figure 8.16: The eGuide database: query interface

functionality is supported. We begin by a brief description of the system architecture that outlines the function of each module. We then continue with our extended example from the previous section, using it to describe how these modules interact with each other.

Figure 8.15 depicts the architecture of the C^3 system. (This figure is a detailed version of Figure 3.1 in Chapter 3 and Figure 7.10 in Chapter 7.) The three central modules of the system are TDiff, CORE, and QSS. QSS acts as a driver for the entire system and supports subscriptions by using TDiff to compute changes and CORE to store and query them. TDiff and CORE each support the *Graphical User Interfaces (GUIs)* described in Section 8.1. They also support the system interfaces described below. QSS consists of a server module that implements the main change notification functionality. To provide a variety of notification mechanisms (e.g., email, alerts), this server can interact with different kinds of *query subscription clients (QSCs)*. The QSS server uses a private *subscription store* as a repository for subscription data such as user-name, password, source database, and other details described below. Recall, from Chapter 7, that our implementation of CORE is based on an encoding and translation scheme that uses the *Lore* semistructured database system as a back-end database. That is, we encode DOEM databases in OEM, and store the encoded databases in Lore. We evaluate Chorel queries by translating them to equivalent Lorel queries on the encoded OEM databases, evaluate these Lorel queries using Lore, and translate the OEM-encoded results back to DOEM.

Recall, from Chapter 3, that all C^3 interactions with the *source databases* are through Tsimmis *wrappers* and *mediators* that support a set of *Lorel query templates*, and return results in OEM. Wrapper and mediator implementation techniques are not a focus of this dissertation and we do not discuss such details here. Further, for simplicity of exposition, we assume in this chapter that we are interacting with a Tsimmis wrapper. However, since the interfaces offered by Tsimmis wrappers and mediators are identical, our system also allows access to one or more source databases through a mediator. We have implemented and used wrappers that allow our system to interface with a variety of source databases such as relational databases, proprietary bibliographic systems, and Web databases. For example, consider the eGuide

Web site introduced in the previous section, depicted in Figure 8.4. This Web site also supports a simple query interface that allows one to search for movies based on titles, keywords, and a few other criteria, as suggested by Figure 8.16. The wrapper we have implemented for the eGuide database translates Lorel queries that match a supported query template into queries using this form interface. The query result presented by eGuide, in a format similar to that suggested by Figures 8.4 and 8.5, is translated by the wrapper into an OEM format with subobject structure reflecting the hierarchical structure suggested by Figures 8.4 and 8.5. The techniques used by the wrapper to implement such functionality are described in [PGGMU95, HGMC+97, HBGM+97]. Briefly, for each query template that is supported by the wrapper, we indicate the corresponding source query to be sent to the Web site's search interface. This translation of queries is specified using pattern matching with placeholder variables. The set of interlinked Web pages returned as query result by the Web site is converted into OEM using a powerful navigation and pattern-matching language very similar to Perl [WCS96].

We now describe the functioning of the rest of the C^3 system using our example subscription NR-titles from the previous section. It may be helpful to refer to the description of subscription semantics in Section 7.7 of Chapter 7. As soon as the subscription creation request is submitted, QSS records the subscription data, such as the user name, the source database, the polling and filter queries, and the corresponding frequencies in a private *subscription data store*.

8.2.1 Polling

At each *polling time* specified by the polling frequency of the subscription, including the implicit polling time that is the subscription creation time, QSS sends the polling query to the wrapper for the source database. For our continuing example, the following Lorel query is sent to the eGuide wrapper.

```
select movie where movie.rating = "NR";
```

This wrapper finds a match between this query and the following template in its translation database, which uses \$1 as a placeholder variable.

```
    select movie where movie.rating = "$1";
```

The corresponding source query template in the wrapper's translation database indicates a form (see Figure 8.16) with the rating radio button set to the value of the variable \$1, which is "NR" in our example. The resulting form is submitted to the eGuide database, which responds with a collection of interlinked Web pages similar to that suggested by Figures 8.4 and 8.5. Using pattern-matching with variable binding, this collection of pages is transformed to the graph-based OEM format.

We use *OEM load files* to encode OEM data throughout the C^3 system. Figure 8.17 depicts an excerpt of the OEM load file generated by our eGuide wrapper. Recall that OEM data is modeled as a rooted graph O. In order to represent this rooted graph in the linear format required by a load file, we first select an arbitrary rooted spanning tree T of O such that T and O have the same node as root. The tree T is encoded in the load file as follows. Each line in the file represents one node, and consists of the following fields in order: (1) one plus the depth of the node; (2) the label of the tree arc leading to the node; (3) the type of the node; and (4) the value of the node, if the node is of atomic type. For example, the second line in Figure 8.17 encodes a node with incoming tree arc labeled "movie." This node is at depth 1 in the tree, and has children labeled title, category, rating, runtime, and so on. (In the figure, lines beginning with the "+" represent long lines that have been wrapped for presentation purposes.) Details of an enhanced and extensible version of our load file format are described in [GCCM96]. We use this extended format to represent DOEM data in load files by encoding annotations on a node at the end of the line representing the node, and annotations on an arc at the end of the line representing the target node.

The OEM load file thus generated is sent to the TDiff module along with the saved load file from the previous polling time. (At the first polling time, the role of the saved file is played by a trivial OEM load file representing an empty database.) TDiff parses these load files and computes a set of changes describing the difference between them, using our change detection techniques from earlier chapters. Since our source database is text-based, we treat all values as strings and use the scaled character frequency histogram difference function described in Chapter 5 to compare

```
1   ViewRoot   complex
2      movie   complex
3         title   string   The Opposite of Sex
3         category   string   Comedy
3         rating   string   R
3         runtime   string   01:40
3         littleman_rating   string   Little Man Clapping
3         review   string   (At the Embarcadero Center Cinema) A likable cast
+ makes this extremely quirky comedy fun to watch despite a story that strains
+ credibility. Christina Ricci is a riot as a Lolita wannabe who seduces her gay
+ brother's boyfriend. Lisa Kudrow is just as funny in the less showy role of an
+ uptight schoolteacher. The movie meanders a lot, but the kooky characters
+ compel you to stay with it. Directed by Don Roos.
3         full_review   complex
4            title   string   Ricci Plays the Bad Girl With Abandon in
+ 'Opposite'
4            author   string   Ruthe Stein, Chronicle Staff Critic
3         cinema   complex
4            name   string   Palo Alto Square
4            time   string   4:30-9:30
4            phone   string   (650)32M-OVIE
4            address   string   Corner of Page Mill Road and El Camino
+ Real, Palo Alto
3         cinema   complex
4            name   string   Towne 3
4            time   string   5:00-9:35
4            phone   string   (408)287-1433
4            address   string   1433 The Alameda, San Jose
...[some material skipped]...
2      movie   complex
3         title   string   Smoke Signals
3         category   string   Comedy
3         rating   string   PG-13
3         runtime   string   01:29
...[truncated]...
```

Figure 8.17: An OEM load file

two strings. For other datasets, we use different types such as integers and floating point numbers.

Recall that our differencing algorithms in earlier chapters are for trees, not for the general graphs encountered in OEM data. Further, the edit operations used by TDiff are tree edit operations such as subtree moves that do not have a counterpart in a more general graph. However, the following two observations allow us to use these algorithms for OEM data: First, although OEM data is in general not tree structured, the data generated using Tsimmis wrappers is either tree structured or has a preferred spanning tree. For example, most Web sites are intuitively mapped to graphs that are not trees. Several Web pages may point to a common page, and there are often cycles of Web links. However, in spite of such non-tree features, most Web databases also have a preferred spanning tree that reflects their hierarchical structure. In particular, the eGuide database from our running example has a clear hierarchical structure. Part of this structure is readily apparent in Figure 8.4. Second, the tree edit operations are easily mapped to the OEM change operations described in Chapter 7. For example, a subtree move is mapped to one arc addition and one arc removal. In our implementation, the preferred spanning tree of an OEM graph is implicit in its linearization as an OEM load file. That is, an OEM load file linearizes a graph based on the preferred spanning tree; all non-tree arcs are represented using symbolic references to other nodes and are easily detected during parsing [GCCM96].

The OEM changes thus computed by TDiff are presented in an *incremental DOEM load file*, which can be thought of as a simple encoding of an edit script. Figure 8.18 depicts an excerpt from such an incremental load file, one used to load the data displayed in Figure 8.13 on the 11th of September. Each line in an incremental load file is a record representing an edit operation, with fields separated using the ":" character. The first field is always the type of the edit operation. The number and kind of the following fields depend on the type of edit operation the line represents. (In Figure 8.18, lines beginning with the "+" character are wrapped lines continuing the previous ones.) For a node creation operation, denoted by CRE, the second, third, and optional fourth fields represent, respectively, the identifier, creation timestamp, and optional value of the created node. As indicated by Figure 8.18, node identifiers are

```
CRE:    tdiff42540 :    Fri Sep 11 00:00:15 PDT 1998
ADD:    ViewRoot,       tdiff42540,     movie : Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42541 :    Fri Sep 11 00:00:15 PDT 1998,   "Buffalo 66"
ADD:    tdiff42540,     tdiff42541,     title : Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42542 :    Fri Sep 11 00:00:15 PDT 1998,   "Drama"
ADD:    tdiff42540,     tdiff42542,     category :      Fri Sep 11 00:00:15 PDT 1998
ADD:    tdiff42540,     tdiff24192,     rating :        Fri Sep 11 00:00:15 PDT 1998
REM:    tdiff42468,     tdiff24192,     rating :        Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42544 :    Fri Sep 11 00:00:15 PDT 1998,   "01:50"
ADD:    tdiff42540,     tdiff42544,     runtime :       Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42545 :    Fri Sep 11 00:00:15 PDT 1998,   "1 Stars"
ADD:    tdiff42540,     tdiff42545,     star_rating :   Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42546 :    Fri Sep 11 00:00:15 PDT 1998,   "First-time director Vincent
+ Gallo stars in this self-consciously stylized story of a social misfit who's
+ rescued from emotional annihilation by the girl he kidnaps. With Christina Ricci,
+ Ben Gazzara, Anjelica Huston. (profanity, bloody images)."
ADD:    tdiff42540,     tdiff42546,     review :        Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42547 :    Fri Sep 11 00:00:15 PDT 1998
ADD:    tdiff42540,     tdiff42547,     full_review :   Fri Sep 11 00:00:15 PDT 1998
CRE:    tdiff42548 :    Fri Sep 11 00:00:15 PDT 1998,   "Gallo Scores With `Buffalo
+ '66'/Gritty, comic look at forgiveness"
```

Figure 8.18: An incremental DOEM load file

represented as strings. These strings can be thought of as external object identifiers of objects in the DOEM database stored in CORE. Although these identifiers can be implemented using Lore *names* [MAG+97], for greater efficiency they are implemented using a separate symbolic reference index. For a node update operation, denoted by UPD, the second, third, and fourth fields represent, respectively, the identifier, update timestamp, and new value of the node. For an arc addition operation, denoted by ADD, the second, third, fourth, and fifth fields represent, respectively, the identifier, source node identifier, target node identifier, arc label, and addition timestamp of the new arc. For an arc removal operation, denoted by REM, fields are analogous to those for ADD.

The incremental DOEM load file produced by TDiff is then sent to the CORE module for incremental loading into the DOEM database of the subscription being serviced. Refer back to Figure 7.6 in Chapter 7 for the internal architecture of the CORE module. In our example, the edit operations specified by the incremental load file suggested by Figure 8.18 are applied to the DOEM database of the subscription NR-titles.

8.2.2 Filtering and Browsing

At each *filter time* specified by the filter frequency of the subscription being serviced, including the implicit filter time that immediately follows the initial polling time (subscription creation time), QSS first replaces any special variables of the form `t[-i]` in the query with the appropriate time constants based on the current time and the stored past filter times for the subscription. (See Chapter 7 for the semantics of such replacements.) For our continuing example, suppose the previous filter time is midnight on the 11th of September. Then replacing the `t[-1]` in the filter query from the previous section with this timestamp gives us the following Chorel query. (We use a timestamp format similar to that used by the Unix *date* utility.)

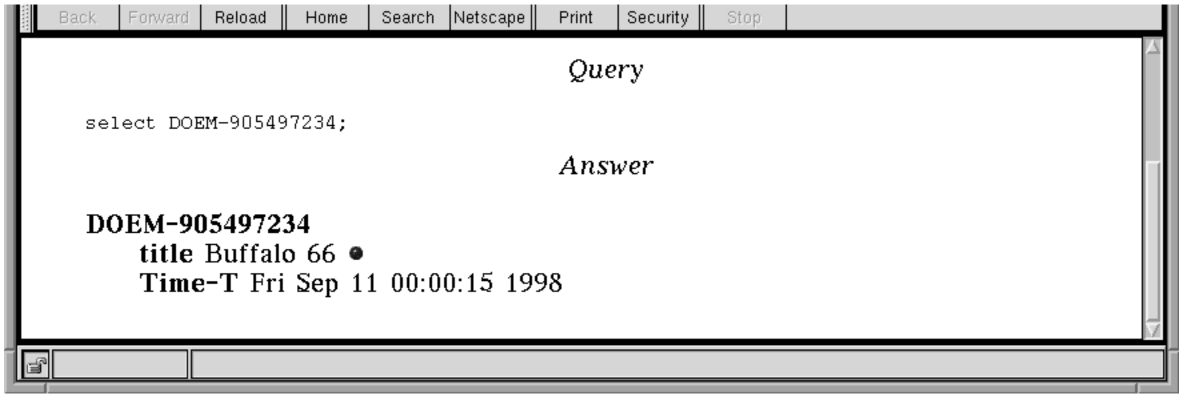
```
select X,T
from ViewRoot.<add at T>%.title X
where T > Thu Sep 11 00:00:00 1998;
```

This Chorel query is sent to CORE, which evaluates it over the DOEM database of the subscription NR-titles (using Lore and the translation scheme described in Chapter 7).

For our example subscription NR-titles, the filter frequency is tied to the polling frequency, implying that each filter time immediately follows the completion of all actions required at each polling time. Recall that at the last such polling time, the changes encoded by the incremental load file suggested by Figure 8.18 were incorporated into the DOEM database for NR-titles. Thus the newly added details for the movie "Buffalo 66" satisfy the query with `X = "Buffalo 66"` and `T = Fri Sep 11 00:00:00 1998`.

If the Chorel query result produced by the CORE module is a nonempty set of object identifiers, this set needs to be stored for future browsing by the subscription owner. To store the result set, we create a new *named* complex object, called the *result object*, that has the objects in the result as subobjects. Recall from Chapter 7 that named objects are points of entry into, and roots of persistence of, Lore and CORE databases. At each filter time at which a nonempty filter query result is produced, a unique name is generated for the complex object used to store the result in this manner, and this mapping between filter times and result object names is maintained by QSS using the subscription store. Finally, if the subscription owner has requested notification of new results using email or other methods, a suitable message indicating the subscription name and filter time is generated and sent.

When a subscription owner visits the subscription review screen depicted in Figure 8.10 and selects one of the change query results for browsing, QSS maps the timestamp of that result to the name of the corresponding result object as discussed above. Continuing our example of the subscription NR-titles, suppose the subscription owner requests the change query result dated September 11th. Using the information in the subscription store, QSS maps this timestamp to the name `DOEM-905497234`. Next, QSS retrieves the result object with this name using a simple Chorel query. In our example, the query is `select DOEM-905497234;`. The resulting CORE *browsing screen*, depicted in Figure 8.19, displays the browsable results of the subscription on the requested date.



Query

```
select DOEM-905497234;
```

Answer

DOEM-905497234
 title Buffalo 66 ●
 Time-T Fri Sep 11 00:00:15 1998

Figure 8.19: Browsing a filter query result

Using named result objects to store and retrieve past results of a subscription as described above raises a subtle issue related to historical accuracy of the subsequent browsing. Since the DOEM database of a subscription continues to evolve as new changes detected by QSS and TDiff are incorporated, the subobject structure visible while browsing a filter query result using stored result objects (such as the object named `DOEM-905497234` in our example) is in general different from that at the time the result was produced. For example, suppose the title displayed in Figure 8.19 is updated to "Buffalo Strikes Back" on the 15th of September, and the owner of our example subscription NR-titles browses the result of 11th September on the 17th. Instead of the proper result displayed in Figure 8.19, the result would contain the new title "Buffalo Strikes Back." Other changes to the DOEM database can affect the result in more subtle ways. For example, if the movie "Buffalo 66" received an R rating on September 14th, such a rating would be visible during browsing the result depicted in Figure 8.19, resulting in an apparent inconsistency. (Recall that the NR-titles subscription asks for only NR-rated movies.) To avoid such inconsistencies, we use the timestamp-restricted browsing introduced earlier. In addition to retrieving the named result object using a query such as `select DOEM-905497234;`, we also restrict the subsequent browsing to the interval $[t_0, t_r]$, where t_0 is the subscription creation time, and t_r is the timestamp of the result being browsed (11th September in our ongoing example). As a result, any changes made after the result was computed are invisible, and the state of the DOEM database as presented by the browsing interface is identical to that at time t_r.

8.3 Summary

In this chapter, we described the design, implementation, and use of the C^3 system for managing change in heterogeneous, autonomous databases. We described the facilities provided by C^3, and demonstrated their use using an extended example based on a popular Web database of entertainment listings. The TDiff component allows us to compare two versions of a portions of such databases, and to browse the changes between them using an intuitive graphical user interface. The CORE component allows us to store, query, and browse a collection of changes gathered over an extended period of time. Finally, the QSS component implements a powerful subscription language that allows us to monitor interesting changes in heterogeneous, autonomous databases.

We described how the C^3 system supports the above functionality using the techniques from earlier chapters and sister projects such as Tsimmis and Lore. We use template-based Tsimmis wrappers to contain the heterogeneity of our source databases by translating data and queries to and from our integrating model, OEM, and query language, Lorel. We use Tsimmis mediators to integrate data from multiple wrappers and mediators. We use our tree differencing algorithms from Chapters 4, 5, and 6 to detect changes in the source databases. Such changes may be directly browsed or stored in the CORE database system described in Chapter 7. The CORE system also supports Chorel queries over the history of a database. The implementation of CORE uses the Lore database management system for semistructured data. The QSS module implements subscriptions by periodically querying Tsimmis wrappers, computing new changes using TDiff, and computing new subscription results using CORE.

We have found the C^3 system to be a valuable tool for managing the complexity of evolving heterogeneous, autonomous databases. For example, the author makes regular use of a QSS subscription similar to the one described in Section 8.1.2 to be notified of interesting movies playing in one of the few small theaters close to Stanford. Using CORE's browsing and querying interface, it is interesting to study the difference between the kinds of movies that play in these theaters and those that play in the large cineplexes. We have also found QSS subscriptions on the KRON traffic reports Web site to be very useful [KRO98]. For example, it is simple to set up a subscription that sends a notification whenever there are accidents on Highway 280 between Stanford and San Francisco on Friday evenings. Further, the ability to browse and query accidents and other traffic events from the past is often useful. For example, on receiving a notification indicating three accidents on Highway 280 one Friday evening, we can query CORE to find accidents on similar evenings in the past year to determine whether the current situation is substantially different and thus a cause for concern.

The stand-alone interface to the TDiff module is also interesting to use. In fact, one of the most entertaining applications of the C^3 system has been the study of how restaurant reviews from the Palo Alto Weekly evolve over time [PAW98]. We list below a few of the actual changes in the restaurant reviews as detected by TDiff. Note that due to numerous moves and other structural changes, simpler differencing algorithms based on computing a longest common subsequence as discussed in Chapter 2 are unable to detect the above changes accurately. These changes to the reviews reveal far more than the reviews themselves, supporting our claim that changes to data are often much more interesting than the data itself.

- The sentence "The kitchen just doesn't make technical errors" in the review of a prestigious restaurant was changed to "The kitchen rarely makes technical errors."

- "In general, the food here is middling to good, but as long as you order carefully, you'll do fine" was deleted.

- "Portions are ladylike and the menu is a bit pricey" was changed to "Portions are ladylike."

- “The only complaint here is the emphasis on meat; many entrees haven’t a hint of green vegetables” was deleted.
- “The prices are moderate” was changed to “It’s all delicious, but it’s not the cheapest meal in town.”
- “Night life in Palo Alto is nothing to write home about as a rule, but the new and improved [...] makes things brighter” was deleted.
- The remark “(pronounced furr)” was inserted after “...huge servings of pho.”

In conclusion, we believe our C^3 implementation demonstrates both the benefits of a coherent change management system for heterogeneous, autonomous databases, and the feasibility of our techniques for building such a system.

Chapter 9

Experimental Evaluation

In this chapter, we present some experimental evaluation of our tree differencing algorithms described in Chapters 5 and 6. We study the effectiveness of our pruning technique, the quality of the computed differences, the merits of different edge cost estimation functions, and the running time of our implementation. We present results both for real data, obtained from the C^3 system described in Chapter 8, and synthetically generated data. For the experiments described in this chapter, we used the parallel transformation model described in Chapter 6 along with the pruning and cost estimation techniques of Chapter 5. In Section 9.1, we describe the results of our experiments using real data. Section 9.2 presents analogous results for synthetically generated data, focusing on how they differ from the results in Section 9.1. We summarize our results in Section 9.3.

9.1 Experiments Using Real Data

Recall the description of the C^3 change management system in Chapter 8. One of the autonomous databases that we used for demonstrating our work is the *eGuide* Web database, which contains hierarchically structured information about entertainment events [EG98]. Our change management system supports subscriptions to changes in this database. Such subscriptions are implemented by periodically querying the eGuide database and comparing the new and old results using our tree differencing algorithm. For our experiments, we used data from a subscription over the portion of the database describing movies. We gathered a sequence of 151 snapshots of this data over a period of five months. We then used the 150 pairs of successive snapshots as inputs to our algorithm. Figure 8.17 in Chapter 8 depicts one of the sample inputs used in our experiments. The input format lists one object per line, with fields suggesting the depth, label, type, and optional value of the object. (In Figure 8.17, long lines, such as the eighth, are wrapped for presentation purposes. For details on the load file format, refer to Section 8.2.1 in Chapter 8.)

Recall that our tree differencing algorithm uses an arbitrary, domain-dependent function to compare node labels. Given two labels, this function returns the cost of updating one to the other. The eGuide dataset consists almost exclusively of string labels. To compare strings, we use the character frequency histogram difference function described in Section 5.6 of Chapter 5. Recall that this function is the scaled sum, over all characters c, of the unsigned difference in the frequencies of c in the two strings. We call the scaling factor *tick*; a higher tick value results in stricter comparisons. For example, with tick $= 0.1$, the cost of updating the string "foo" to the string "fooos" is $0.1 \times (|1-1| + |2-3| + |0-1|) = 0.2$. In our experiments below, we study the effect of varying tick.

All the studies described in this section were performed on all 150 pairs of input trees, and in all the charts presented in this section, except Figures 9.7 and 9.8, each data point is the average result over these 150 trials.

9.1.1 Effectiveness of Pruning

As described in Chapter 5, pruning the induced graph is a very important step in our method for comparing trees. In addition to exponentially reducing the size of the search space for min-cost edge covers of the induced graph, better pruning also results in better initial solutions, as described in Section 9.1.2 below. Recall, from Chapter 5, that we prune an edge e from the induced graph if the lower bound cost of e is greater than or equal to some quantity Q; that is, we prune if $c_{lb}(e) \geq Q$. Such pruning is conservative; that is, the pruned induced graph is guaranteed to contain an edge cover no more expensive than any min-cost edge cover of the unpruned induced graph. In Chapter 5 we conjectured that our pruning rules are excessively conservative in most situations and suggested that it may be profitable to prune more aggressively. To test this conjecture, we studied the effect of varying an *aggressiveness parameter* A, where we now prune an edge e if $c(e) \geq Q(1-A)$. Thus $A = 0$ corresponds to conservative pruning, $A = -\infty$ corresponds to no pruning, and values of A approaching 1 correspond to very aggressive pruning.

We quantify the success of pruning using the *excess edge ratio* defined as follows, where $|I_p|$ is the number of edges in the induced graph after pruning, and $|I|$ is the number of edges before pruning.

$$eer = \frac{|I_p| - (min(|T_1|, |T_2|) + 1)}{|I|}$$

Note that an edge cover of the induced graph of trees T_1 and T_2 contains at least $min(|T_1|, |T_2|) + 1$ edges. The intuition behind eer is that any edges beyond this number are in excess; a pruned induced graph with $eer = 0$ has no excess edges. A pruned induced graph with no excess edges is a minimum-cardinality edge cover of the induced graph. However, in our cost model described in Chapter 5, edges have differing costs. Therefore, a minimum-cardinality edge cover is not, in general, a minimum-cost edge cover. Thus, in general, the pruned induced graph defined by a minimum-cost edge cover has $eer > 0$. Therefore, $eer = 0$ is only a loose lower bound on the number of unnecessary edges remaining in an induced graph. Figure 9.1 indicates how the excess edge ratio varies with pruning aggressiveness for different values of the label discrimination parameter *tick* introduced at the beginning of Section 9.1. As expected, higher values of the aggressiveness parameter A lead to fewer excess edges. We note that even conservative pruning ($A = 0$) results in a more than a 50% reduction in the number of excess edges, indicating that our pruning technique is very effective for this real dataset. Further, as we prune more aggressively, the excess edge ratio continues to drop significantly, approaching a value very close to 0. Figure 9.1 also shows that as tick is lowered, pruning is less successful. The reason for this result is that a lower value of tick results in node labels appearing more similar to each other, in turn lowering the lower-bound edge costs.

One may expect highly aggressive pruning to lead to a deterioration in the quality of the solution, since with increasing aggressiveness it becomes more likely that the edges required for a good solution are removed. We therefore studied the effect of aggressive pruning on the relative cost of the transformation corresponding to the initial solution produced by our algorithm.

Recall, from Chapter 5, that the cost of a transformation is given by the sum of the costs of its constituent edit operations; the costs of edit operations are based on a parametric cost model described in that chapter. Ideally we would like to compare the cost of a transformation computed by our method to the cost of an optimal (minimum-cost) transformation. However, given the NP-hardness of the problem and the size of our input data, computing such an optimal solution is impracticable. Therefore, to judge the merit of a transformation, we compare its cost to the cost of the best transformation computed for the given inputs by all our experiments. Although in general the best computed transformation is not guaranteed to be optimal, by inspecting several such transformations for our sample data, we found that the best computed transformation is very often optimal or close to optimal.

We define the relative cost of a transformation F as $c(F)/c(F^{*})$, where $c(F)$ denotes the cost of F, and $c(F^{*})$ denotes the cost of the best computed transformation. Figure 9.2 shows that the relative cost of the solution produced decreases as we prune more aggressively up to values of A as high as 0.95. This result is explained by noting that as more edges that are very likely (although not guaranteed) to be undesirable in an edge cover are pruned, the minimum-cost edge cover computed in the next step of our method is less likely to contain such undesirable edges. In addition, the boundary cases that prevent us from conservatively pruning edges that may be pruned at a higher levels of aggressiveness are typically uncommon; in particular, they are rare in the eGuide dataset used in our experiments here.

For other datasets, the value of the aggressiveness factor A that gives the best results is, in general, different. Using an A value that is too low results in less pruning and a greater chance of making the wrong choices when computing the transformation. On the other hand, an A value that is too high results in too much pruning and

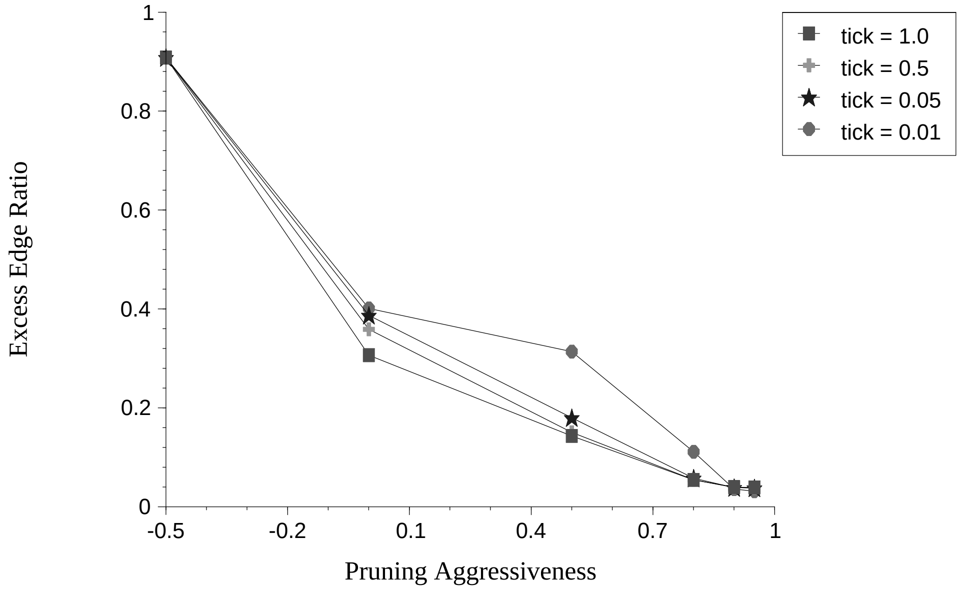


Figure 9.1: Effectiveness of pruning for eGuide data

a greater chance of edges needed for a minimum-cost transformation being removed. It would be prudent to run a few test cases with different values of A to empirically determine a suitable value. Our experience with the datasets used in our implementation described in Chapter 8, as well as the synthetic dataset studied in Section 9.2 indicates that suitable values for A are approximately in the range $[0.8, 0.95]$. Figure 9.2 also shows that lower values of tick lead to poorer results. A lower tick value leads to less pruning (as indicated by Figure 9.1), and also reduces the effectiveness of heuristic functions used to estimate edge costs, as discussed in Section 9.1.2.

9.1.2 Quality and Edge Cost Estimates

Recall from Chapter 5 that, given the hardness of the tree comparison problem, it is not possible to devise a purely edgewise cost function c^* on the edges of the induced graph such that the cost $\Sigma_{e \in K} c^*(e)$ of an edge cover K is the same as the cost of the corresponding transformation unless $\mathcal{P} = \mathcal{NP}$. Our method therefore uses an edge cost estimation function c' that approximates such a function c^*. Such as estimation

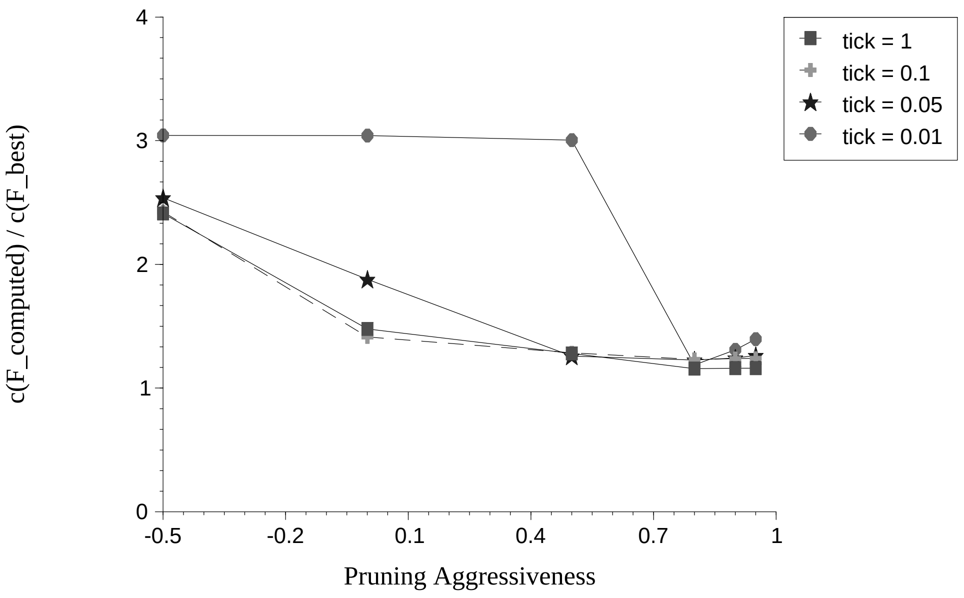


Figure 9.2: Effect of pruning on quality for eGuide data

function is used to compute a minimum-cost edge cover of the induced graph; this edge cover intuitively matches nodes in one input tree to their counterparts in the other. In Chapter 5 we suggested using the lower bound cost of an edge as the estimated cost. However, it is also possible to use other heuristic cost estimates. We experimentally evaluated the following edge cost estimation functions:

1. **LAB:** The estimated cost of an edge $[m, n]$ is the cost of updating the label of m to that of n. That is, $c_1([m, n]) = c_u(m, n)$. This estimate is likely to produce good results when the node labels of the input data are good discriminators of the nodes. In particular, if the node labels constitute keys or object identifiers that are shared between the input trees, this estimate will result in nodes in one tree being matched to their counterparts in the other. However, if the node labels are not good discriminators of the nodes, this estimate is likely to serve as a poor guide for matching nodes.

2. **LAB+SS:** The estimated cost of an edge $[m, n]$ is the sum of the cost of updating the label of m to that of n and the difference in the sizes of the two

subtrees. That is, using $|st(x)|$ to denote the size of the subtree rooted at node x, we have the following:

$$c_2([m,n]) = c_u(m,n) + |\, |st(m)| - |st(n)|\, |$$

Intuitively, our first estimate, LAB, ignores the positions of nodes in their trees. The estimate LAB+SS attempts to address this deficiency of LAB by differentiating nodes using the size their subtrees. This estimate is likely to perform well for datasets consisting of trees that have a natural layering and in which matching nodes in different layers to each other is undesirable. For example, if the input trees are layered in the manner described in Chapter 4, this estimate will strongly discourage matching nodes in different layers by assigning a high cost to such edges (due to the high expected difference between the sizes of subtrees rooted at nodes in different layers). On the other hand, if the input has no such layering property (as is the case for our synthetically generated trees in Section 9.2, this estimate is likely to perform poorly, as it will inordinately penalize the matching of nodes in different positions in the trees.

3. **PARM:** In this estimation function, we use parameters to determine the presence of edit operations that may contribute to the cost of an edge. Recall, from Chapter 5, that one of the major reasons we cannot accurately estimate the contribution of an edge (to the cost of a transformation derived from a minimal edge cover containing that edge) is the following: It is not possible to decide whether the procedure for generating a transformation $F(K)$ corresponding to an edge cover K (described in Chapter 6) generates a move, copy, or glue operation corresponding to a given edge e without knowledge of the other edges in K. If a copy operation is generated corresponding to an edge, the cost of that operations, c_C is charged to that edge. In the PARM estimation function, we estimate this component of the cost of an edge to be $p_C.c_C$, where p_C is a parameter in $[0,1]$ that intuitively indicates the likelihood of a copy operation being generated for an edge. (We use the term likelihood in an informal, and not statistical, sense.) Similarly, we estimate the glue component of the cost of

an edge by $p_G.c_G$, where p_G is a parameter in $[0,1]$ that intuitively indicates the likelihood of a glue operation being generated for an edge.

For move operations, we use a similar idea. However, recall from Section 5.5 of Chapter 5 that it is better to charge the cost of a move not to the edges incident on the moved nodes, but to the edges incident on their parents. In particular, if a node x is moved, every edge incident on its parent $m = p(x)$ is charged $c_M/2|E_K(m)|$, where $E_K(m)$ is the set edge cover edges incident on m. Since we do not know $|E_K(m)|$ when we are computing edge cost estimates, we estimate $|E_K(m)|$ by $|E(m)|.p_C + 1$, where $E(m)$ is the set of edges in the pruned induced graph that are incident on m, and p_C is the parameter introduced above to estimate copy costs. Intuitively, this estimate reflects the notion that if copy operations are more likely, $|E_K(m)|$ is likely to be higher. Thus, if a child x of a node m is moved, an edge $[m, n]$ is charged $c_M/2(|E(m)|.p_C + 1)$. Let us further use a parameter p_M that intuitively indicates the likelihood of a node being moved. Then, since m has $|C(m)|$ children, the estimated cost due to moves for an edge $[m, n]$ is $c_M.p_M.|C(m)|/2(|E(m)|.p_C + 1)$. For an edge $[m, n]$, let us define $f = |\ |C(m)| - |C(n)|\ |$ to be the child mismatch factor. We know that at least f nodes must be moved, deleted, or inserted in any edge cover that matches m to n. Intuitively, the greater the value of f, the more likely it is that some of the children of m and n will be moved. We would like to correct the likelihood of moves p_M used above to reflect this intuition. For this purpose, we use a parameter p'_M that indicates the additional likelihood of one of the f mismatched nodes being moved. The correction to apply to the estimated cost of an edge is then $c_M.p'_M.f/2(|E(m)|.p_C + 1)$.

The above argument for the children of m can be repeated for the children of n, giving the following formula for the estimated cost of an edge $[m, n]$ due to moves.

$$\begin{aligned} c_{em}([m,n]) &= \frac{c_M.p'_M.|\ |C(m)| - |C(n)|\ | + c_M.p_M.|C(m)|}{2.(|E(m)|.p_C + 1)} \\ &+ \frac{c_M.p'_M.|\ |C(m)| - |C(n)|\ | + c_M.p_M.|C(n)|}{2.(|E(n)|.p_G + 1)} \end{aligned}$$

The first and second term on the right hand side estimate the cost contribution of moving the children of m and n, respectively. Putting the estimated costs due to copies, glues, and moves together, we have the following formula for the estimated cost of an edge:

$$c_3([m,n]) = p_C.c_C + p_G.c_G + c_{em}([m,n])$$

4. **LAB+POS:** The estimated cost of an edge $[m,n]$ is the label update cost plus the weighted sum of the differences between the height, depth, number of children, and number of siblings of m and n. This weighted sum informally characterizes the difference in the positions of the nodes m and n in their respective trees. Intuitively, the greater the difference in the positions of m and n, the more likely it is that matching m to n will require a large number of edit operations in the resulting transformation (in order to change the position of m in T_1 so that the transformed tree is isomorphic to T_2). More precisely, the estimated cost is given by the following, where w_h, w_d, w_c, and w_s are parameters, and where $h(x)$, $d(x)$, $C(x)$, and $S(x)$ denote, respectively, the height, depth, set of children, and set of siblings of node x:

$$\begin{aligned} c_4([m,n]) &= c_u(m,n) \\ &+ w_h.|h(m) - h(n)| + w_d.|d(m) - d(n)| \\ &+ w_c.|\,|C(m)| - |C(n)|\,| + w_s.|\,|S(m)| - |S(n)|\,| \end{aligned}$$

5. **LB:** The cost of an edge is estimated using the lower bound derived in Section 5.5.2 of Chapter 5.

In order to evaluate the above edge cost estimation functions, we computed a min-cost edge cover using each of the estimates, and compared the costs of the corresponding transformations with each other. For these experiments, we used the eGuide dataset described at the beginning of Section 9.1. The results are summarized in Figures 9.3, 9.4, 9.5, and 9.6, which plot the relative cost of the computed transformation against the aggressiveness of pruning for the five estimation functions,

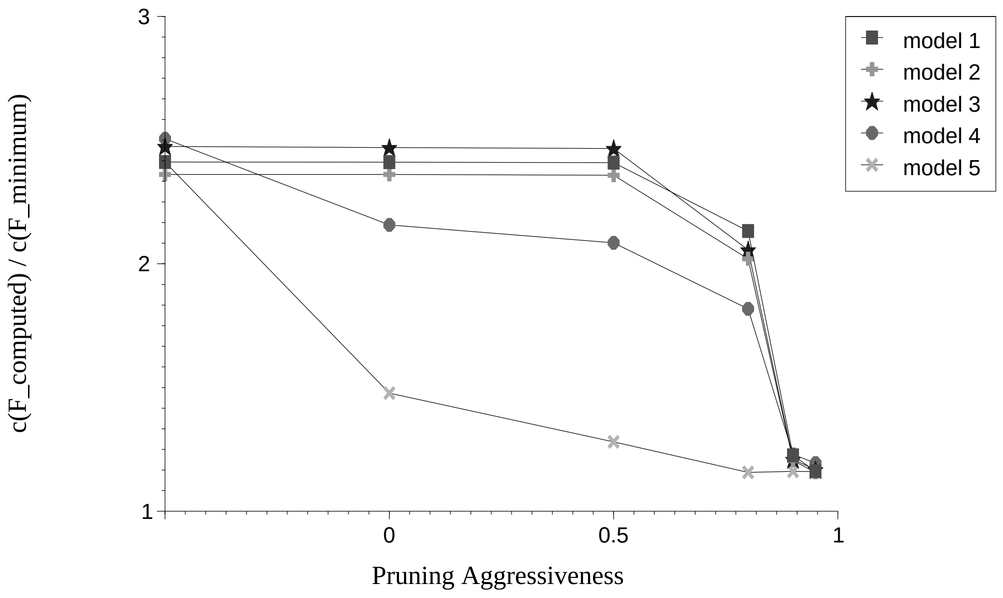


Figure 9.3: Comparison of edge cost estimation methods; tick = 1

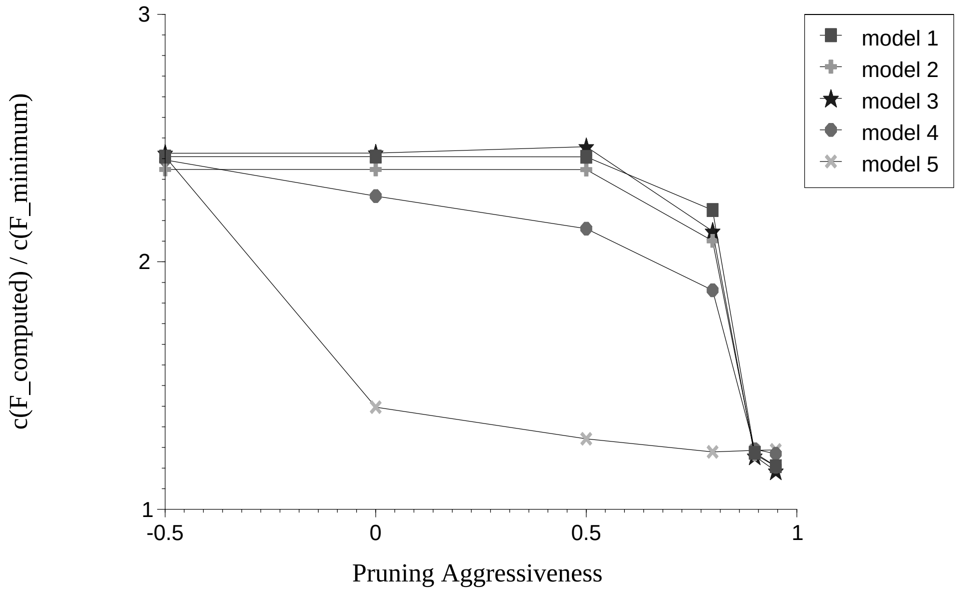


Figure 9.4: Comparison of edge cost estimation methods; tick = 0.1

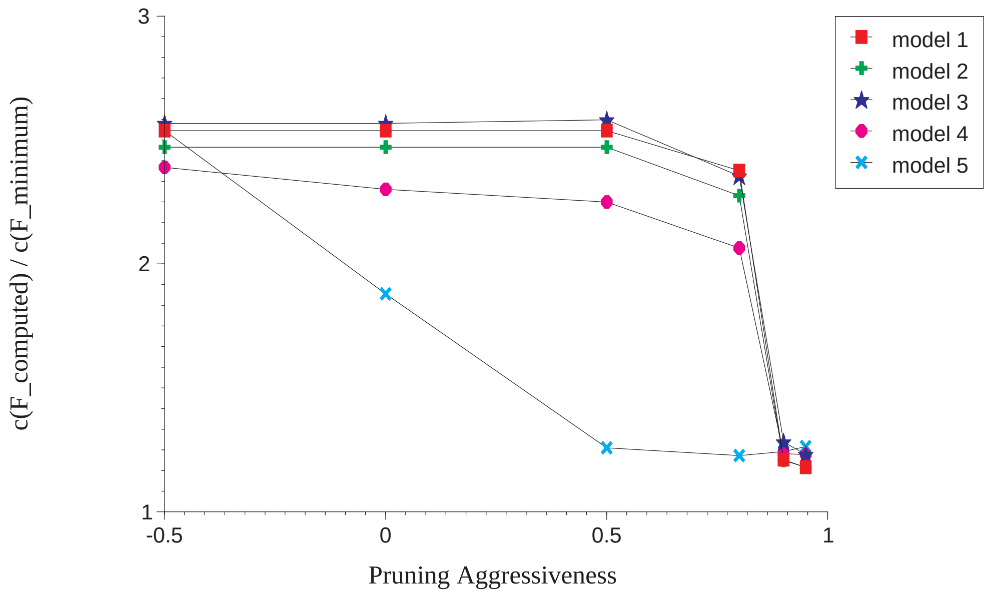


Figure 9.5: Comparison of edge cost estimation methods; tick = 0.05

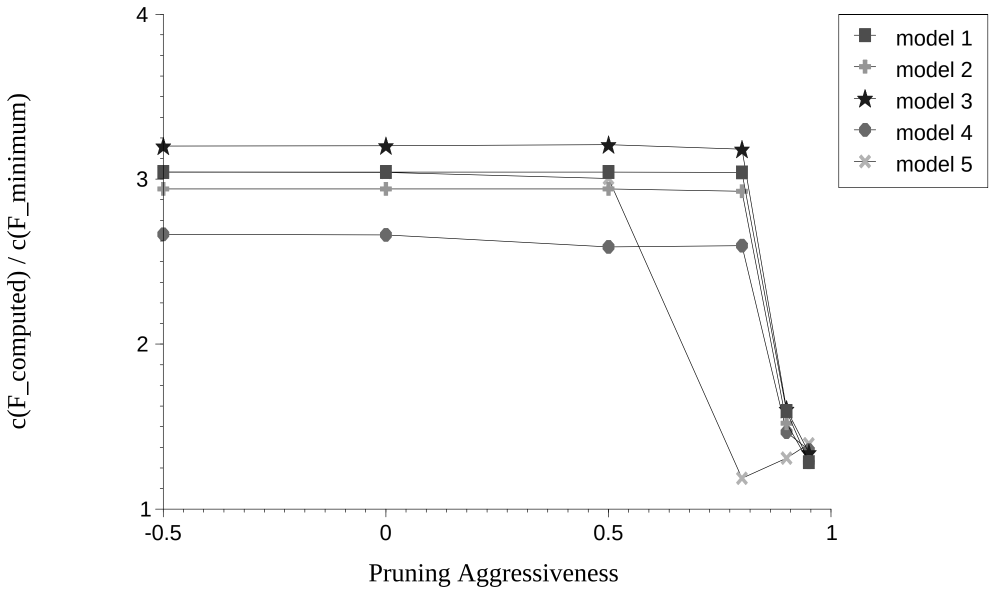


Figure 9.6: Comparison of edge cost estimation methods; tick = 0.01

and for different values of the label discrimination parameter tick. We use a dummy aggressiveness value of -0.5 for experiments in which no pruning is performed.

For all five edge cost estimation functions, more aggressive pruning improves the quality of the computed solution up to very high values of the aggressiveness parameter. This result is consistent with our results of the previous section, which used the LB estimation function. A related observation is that when no pruning is performed, all five estimation functions perform almost equally poorly for tick values of 1, 0.1, and 0.05. The reason for this behavior is that the more sophisticated estimation functions (such as the lower bound estimate) rely on the absence of edges in the induced graph. For example, if no edges have been pruned then LB degenerates to LAB. For tick values of 1, 0.1, and 0.05, LB consistently and significantly outperforms all the others. The result is explained by noting that when the number of edit operations is relatively small, as is the case for much real data in general, and our sample dataset in particular, the best-case scenario assumed by LB is close to accurate. The estimate LAB+POS is the next best performer. When tick is 0.01 we observe that this estimate gives the best results. The reason for this behavior is that at a very low tick value node labels become irrelevant for the purpose of matching nodes because any label can be updated to any other label at a very low cost. (Recall that for our dataset node labels are strings. With tick at 0.01, changing 100 characters in a string costs only 1 unit.) Thus it is more prudent to match nodes giving weight to structural properties. Focusing on the results for the lower bound cost estimate, we note that as tick decreases, the aggressiveness of pruning required to achieve a given quality increases. For example, when tick is 1 or 0.1, conservative pruning (aggressiveness 0) gives us a relative cost of roughly 1.4. At tick values of 0.05 and 0.01, this number deteriorates to approximately 1.9 and 3.0, respectively. The inflection moves from aggression 0 for tick 1 and 0.1 to aggression 0.5 and 0.8 for tick 0.05 and 0.01 respectively. Thus by using a better edge cost estimate, we can attain a higher quality solution at lower levels of pruning aggressiveness.

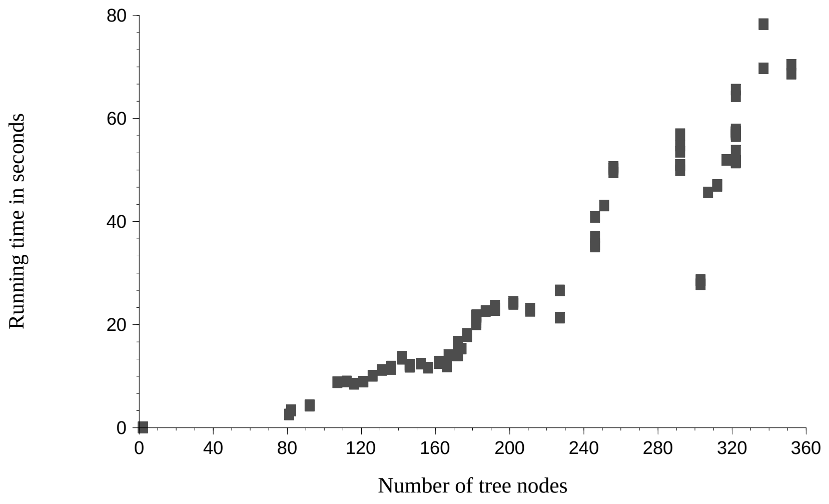


Figure 9.7: Running time for eGuide data

9.1.3 Running Time

Figure 9.7 depicts the effect of input size on running time, where we define input size to be the sum of the number of nodes in the two input trees. These results are for experiments using the LB edge cost estimation function, with a pruning aggressiveness of 0.9 and a tick value of 0.1. We observe that the running time is roughly quadratic in the size of the input. To verify this relationship, Figure 9.8 plots the running time against the product of the tree sizes; we note that the relationship is close to linear. Note that any algorithm used for computing a minimum-cost edit script between trees T_1 and T_2 (even using much simpler edit operations) must make at least $|T_1| \times |T_2|$ label-comparisons [Sel77].

Figure 9.9 depicts the break-up of the running time of among the five major steps of the implementation. We observe that over 40% of the running time is spent in the input and parsing. This step involves reading the two input files, in the format depicted in Figure 8.17, and parsing them into internal tree structures. We believe this time can be substantially reduced by using better parsing techniques, but do not explore this issue further here since it is not the focus of this work. The next step, constructing the induced graph, accounts for roughly 17% of the running time. Note

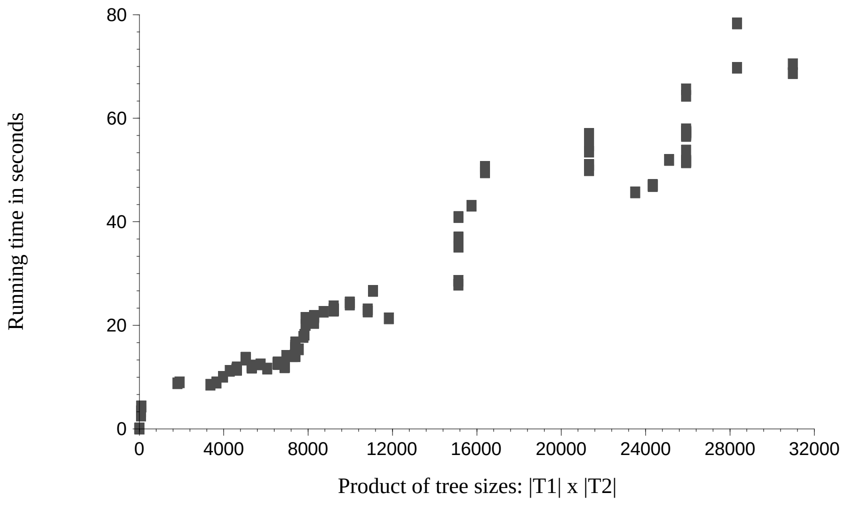


Figure 9.8: Running time for eGuide data

that this time includes not only the time required to build the bipartite induced graph with $O(|T_1|.|T_2|)$ edges, but also the time required to evaluate the user-specified label-comparison function. Recall that in our data labels are strings and the comparison function is based on comparing the character frequency histograms of two strings. Using a simpler and more efficient comparison function can reduce the time spent in this step, but may result is a less accurate result. The pruning step accounts for about 26% of the running time. Note that these numbers are for very aggressive (aggressiveness 0.9) pruning, and we continue pruning until no more edges can be pruned. We can reduce the time spent in this step by stopping the pruning process before it terminates naturally. However, given that fewer edges in the induced graph result in an exponential reduction in the size of the search space for edge covers, such a strategy may result in much higher running times in the search step. Computing a min-cost edge cover using the edge cost estimates accounts for roughly 16% of the running time. Recall that we compute a min-cost edge cover by transforming the problem into a weighted matching problem, and then using a library implementation of Gabow's $O(n^3)$ algorithm for weighted matching in arbitrary graphs [Rot]. By using a more efficient implementation of weighted matching specialized for bipartite

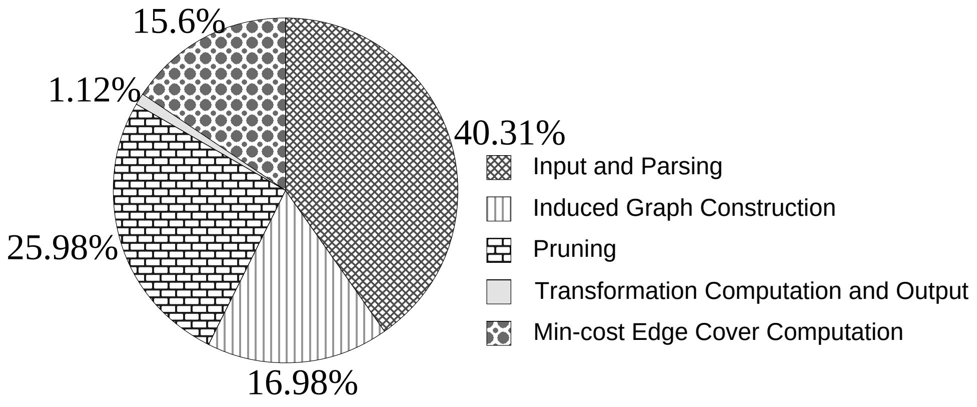


Figure 9.9: Components of total running time for eGuide data

graphs, it is possible to reduce the time spent in this stage. Figure 9.9 also shows that the time spent on computing the transformation from the min-cost edge cover is very small.

9.2 Experiments Using Synthetic Data

In this section, we present the results of our experiments using synthetically generated data, focusing on the differences between these results and the corresponding results for the real data presented above. The results for the running time of our algorithm for synthetic data are essentially identical to those for real data presented in Section 9.1.3. We therefore concentrate on results on the effectiveness of pruning, the quality of the solution produced, and the relative merits of different edge cost estimates.

Generating sufficiently general, yet realistic, random trees to serve as inputs for our experiments is a challenging problem in itself. After experimentation using several input-generation techniques, we decided to use the following inductive method for generating a tree with N nodes. We begin with a tree T_0 containing only the root node. To grow the tree T at any stage we do the following: We randomly select a leaf m of the current tree. We then select an integer c from the uniform random distribution over the interval $[f, F]$ where f and F are parameters representing, respectively, the minimum and maximum fanout of internal nodes. We then add $min(c, N - |T|)$ children to m. We initially assign each node thus generated a unique integer greater than one as label. After we have generated the entire tree, we randomly select $N \times D$ nodes, where D is a parameter in the range $[0, 1]$ denoting the fraction of identical labels. The labels of all these $N \times D$ nodes are then changed to 1. Thus the parameters used to generate a synthetic tree are the following:

- The number of nodes in the tree (N)
- The minimum and maximum fanout of interior nodes (f and F)
- The fraction of nodes that have identical labels (D)

After generating a tree T_1 as described above, we generate a random transformation of the desired size by adding operations one at a time as follows. (For the experiments discussed below, we generated random transformations containing $|T_1|/10$ edit operations.) We first select type of the edit operation, with each type being equally likely. Next, the node in T_1 on which this operation is to act is selected uniformly randomly from the nodes of T_1. Labels of inserted nodes are selected using the method used to select node labels when generating T_1. Labels of updated nodes are generated by adding a uniformly randomly generated delta in the range $(0, 1)$ to the current label of the node being updated. Note that, as a result of such updates, node labels in the second input tree are not integers in general; further, a node's label may closely match that of another. For example, a node with initial label 4 may be updated to 4.99, thus closely matching another node with label 5. We also take special precautions such as making sure the target node of a move operation does not belong to the subtree being moved. Once we have generated a random transformation F_r in this manner, we apply F_r to the tree T_1 go obtain the second input tree $T_2 = F_r(T_1)$. The trees T_1 and T_2 then serve as input to our algorithm.

Since all node labels in the experiments of this section are numbers, we used a simple scaled arithmetic difference to compare node labels. Thus the cost of updating 1.3 to 2 is $0.7 \times t$, where t is the scaling factor. Our experiments showed that the effect of varying t are similar to the effect of varying tick for the experiments in Section 9.1.

The results reported below are for $t = 1$. In the charts in this section, each data point is the average result over at least 15 trials for each of the following values of input tree sizes: 10, 20, 40, 80, and 160, giving at least 75 trials for each data point. We did not find any significant sensitivity of our results, other than running time, to the size of the input trees; therefore we discuss only the aggregated results.

9.2.1 Effectiveness of Pruning

Figure 9.10 illustrates how the effectiveness of pruning varies as we increase the number of nodes that have identical labels. It plots the excess edge ratio defined in Section 9.1.1 against the parameter D described above for two values of the pruning aggressiveness parameter A. We observe that as the fraction of nodes with identical labels rises, the excess edge ratio rises rapidly. This result is expected, since the greater the number of identical labels, the smaller the number of edges with non-zero update costs, leading to smaller values of lower bound edge costs, and thus fewer edges that can be pruned. We also observe that for a given value of D, aggressive pruning ($A = 0.9$) yields a lower excess edge ratio. However, the benefit of aggressive pruning diminishes as D rises. This behavior is explained by noting that as more labels become identical, the lower bound costs of an increasing number of edges drop rapidly to zero; edges with zero lower bound cost are not pruned for any $A < 1$.

Since aggressive pruning is not guaranteed to remove only edges not required by a min-cost solution, one may expect aggressive pruning to result in a deterioration of the quality of the solution. In Section 9.1.1 we observed that for the dataset in our experiments, aggressive pruning actually results in a very significant improvement in the quality of the solution produced. This result indicated that the problem cases that can in general result to a deterioration in solution quality with aggressive pruning rarely occur in our dataset. However, for randomly generated data, such as the data used in our experiments in this section, such problem cases may be expected to occur with a higher frequency. Figure 9.11 illustrates the relation between output quality and the fraction of nodes with identical labels in the input, for both conservative ($A = 0$) and aggressive ($A = 0.9$) pruning. We use the ratio of the cost of the

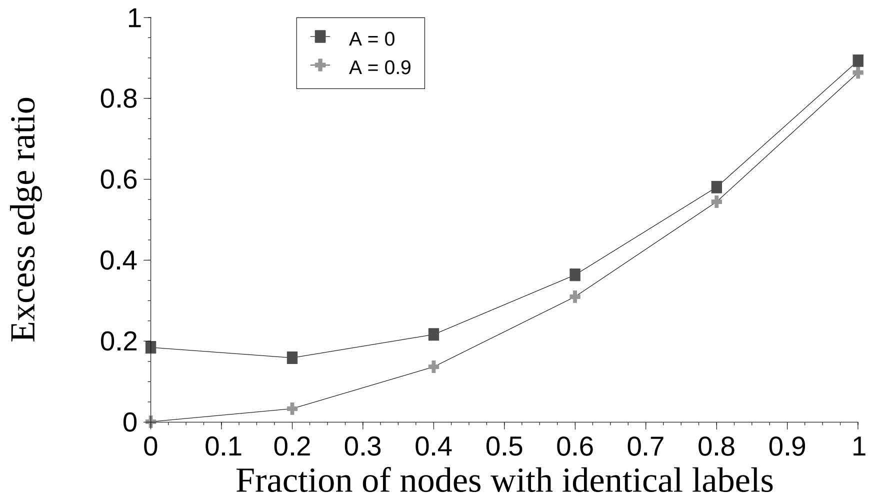


Figure 9.10: Effectiveness of conservative pruning for synthetic data

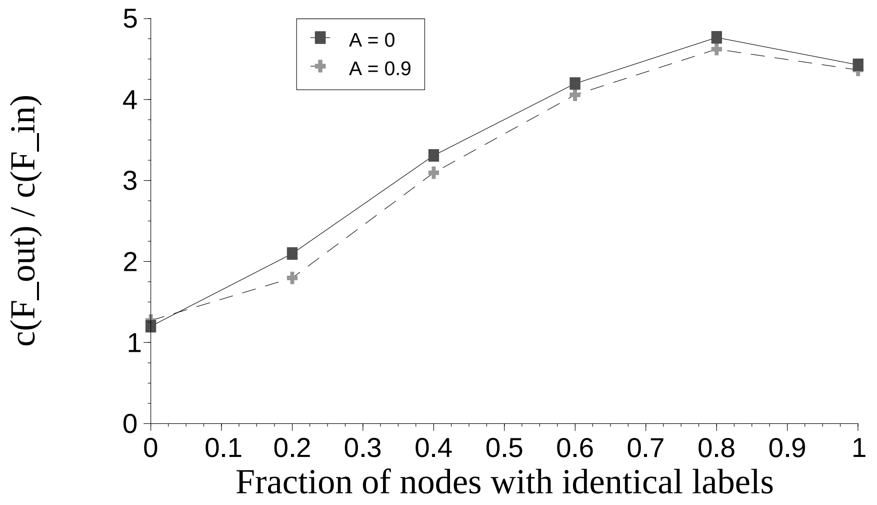


Figure 9.11: Effect of pruning on quality for synthetic data

generated transformation (F_{out}) to the cost of the randomly generated transformation (F_{in}) used to produce T_2 from T_1 as a measure of the inverse of the quality of the solution. We observe that the quality of the solution produced is very sensitive to the fraction of identical labels. This result is not surprising, since as D is increased, our ability to distinguish nodes based on their labels diminishes rapidly. In the extreme case of $D = 1$, node labels are completely useless in determining matching nodes. We also observe that in general aggressive pruning improves the quality. However, this improvement is substantially less than the improvement for real data described in Section 9.1.1. Again, this result is to be expected , since with randomly generated inputs, aggressive pruning is more likely to remove useful edges, thus resulting in worse solutions in some of instances of our synthetic data. Thus when we average over many experiments, the benefits of aggressive pruning in some instances are partially nullified by the penalties in other instances. Finally, note that while the quality of the solution steadily deteriorates as the fraction of identical labels increases, when all labels are identical ($D = 1$), the quality is marginally better than that at $D = 0.8$. This result can be explained by noting that when all labels are identical, the number of potential partners of a node for a low-cost solution increases. For example, if we consider a node $m \in T_1$ at height 1, as D increases, the expected number of nodes $n \in T_2$ such that the subtrees rooted at m and n are isomorphic increases.

9.2.2 Quality and Edge Cost Estimates

We conducted a study similar to the one described in Section 9.1.2 in order to study the relative merits of different edge cost estimates. We computed a min-cost edge cover using each of the estimates, and compared the costs of the corresponding transformations to the cost of the randomly generated input transformation. As in Section 9.1.2, we found that aggressive pruning with aggressiveness A close to 0.9 results in the best results; therefore, we focus on A values in this neighborhood. The results are summarized in Figures 9.12, 9.13, 9.14, 9.15, 9.16, and 9.17, which plot the relative cost of the computed transformation against the aggressiveness of pruning for the five estimation functions, and for different values of D, the fraction of tree nodes with

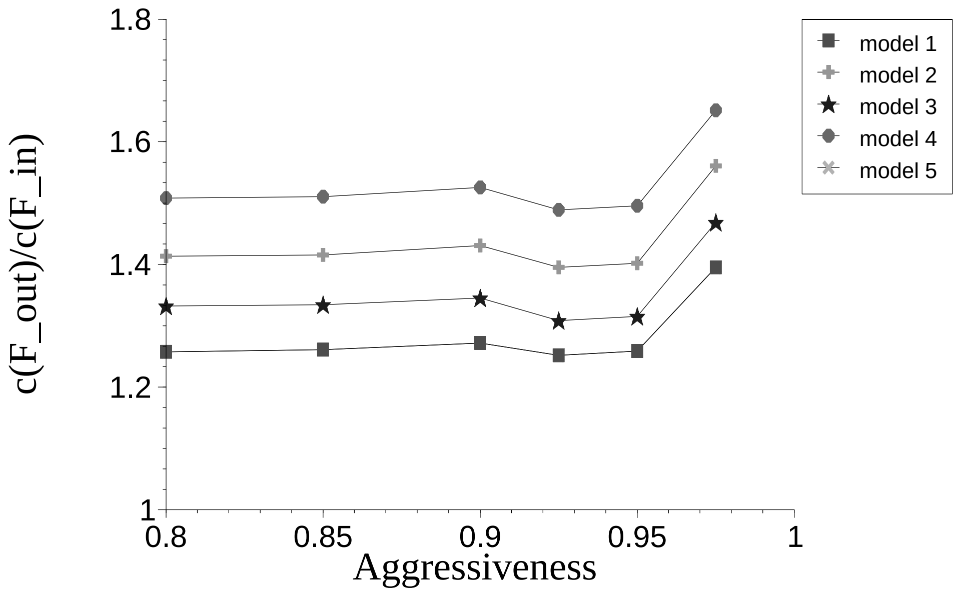


Figure 9.12: Quality and edge cost estimates; $D = 0$

identical labels. Note that increasing D for our synthetic data has an effect similar to decreasing the label discrimination parameter tick for the real data described in Section 9.1.

For the case in which all node labels are distinct ($D = 0$, Figure 9.12), we observe that our first estimation function, LAB, which compares nodes using their labels only, performs the best. This result is not surprising because when all labels are distinct, the labels function almost as keys. The labels are not exactly keys because copy, insert, and update operations may result in the tree T_2 containing multiple nodes with the same label even though T_1 does not. In Figure 9.12, the data points for the LB estimate are superimposed on those for LAB. This result is to be expected because, when most of the node labels differ significantly in their labels, the lower bound cost is dominated by the cost of label update. Edge cost estimates that consider factors other than the label (such as node position and subtree size) perform poorly in this case because they make it more likely for nodes to be mismatched because of these factors.

As soon as there are a few duplicate labels in the input ($D = 0.2$, Figure 9.13),

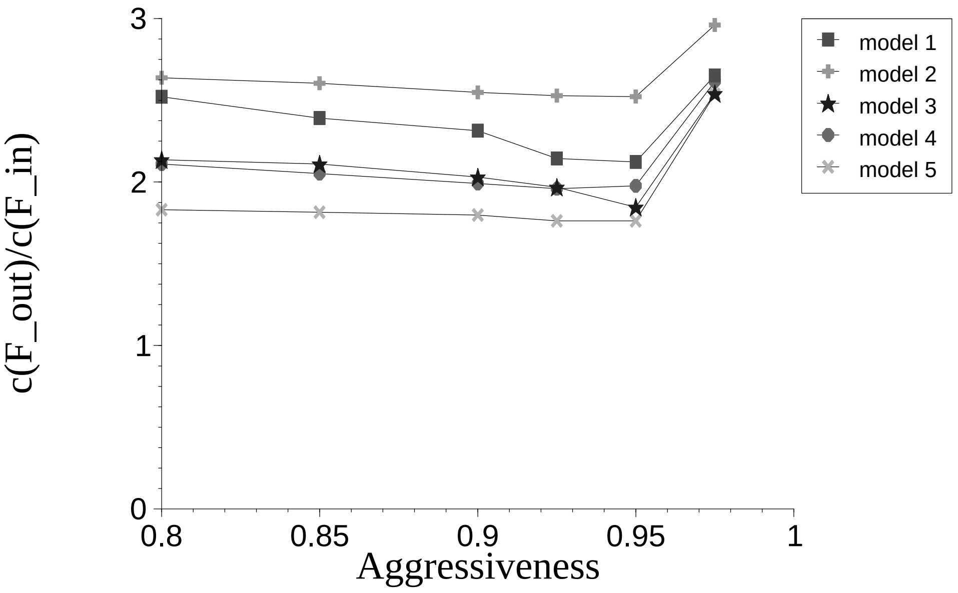


Figure 9.13: Quality and edge cost estimates; $D = 0.2$

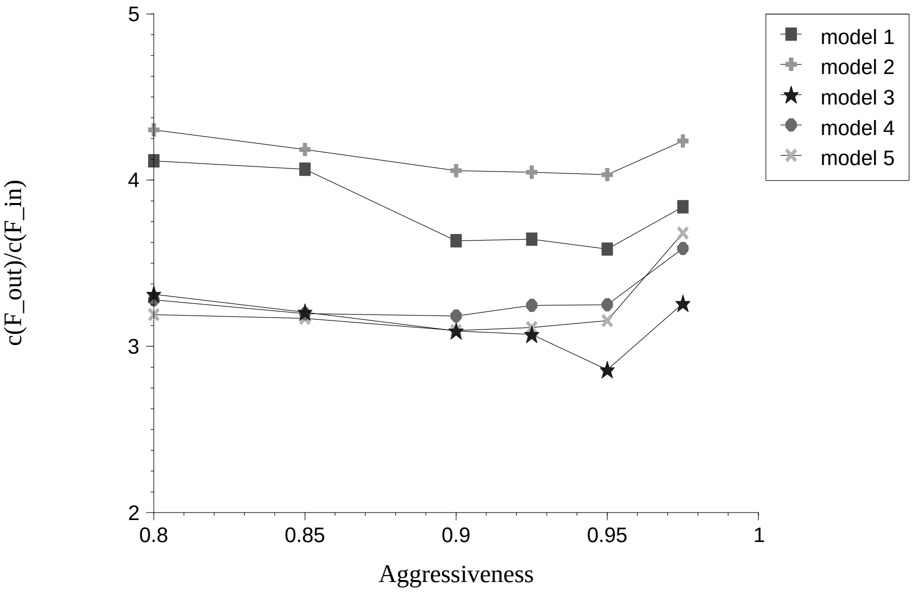


Figure 9.14: Quality and edge cost estimates; $D = 0.4$

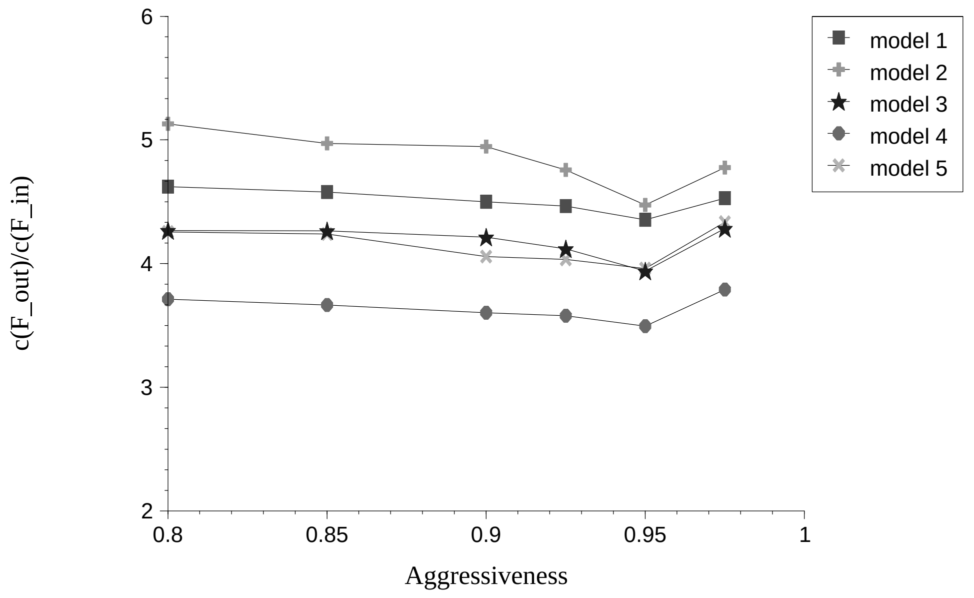


Figure 9.15: Quality and edge cost estimates; $D = 0.6$

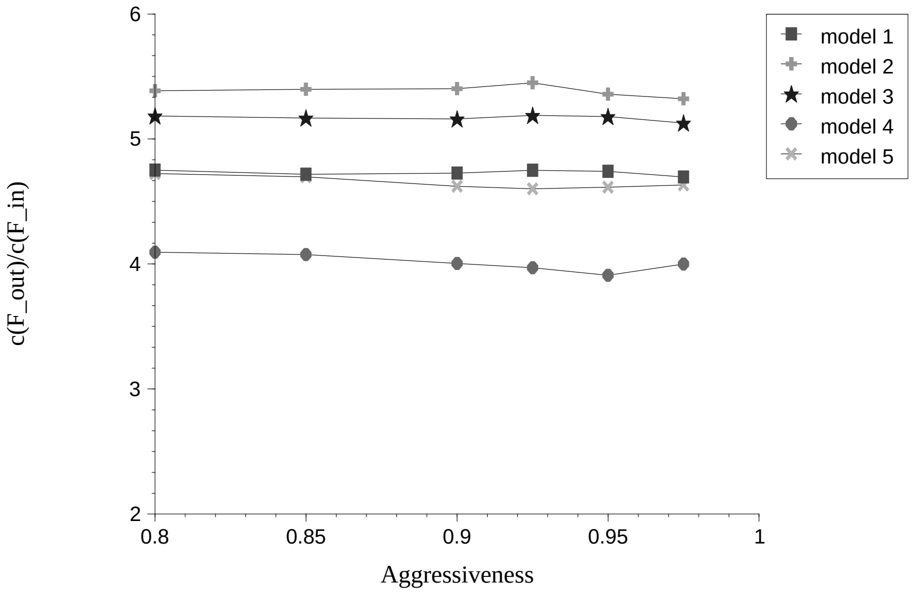


Figure 9.16: Quality and edge cost estimates; $D = 0.8$

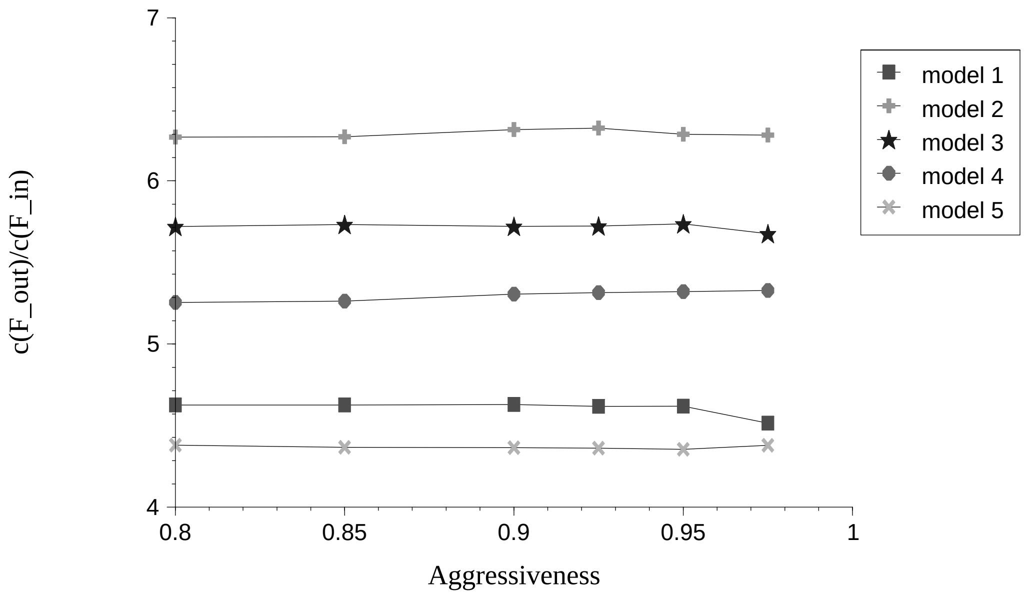


Figure 9.17: Quality and edge cost estimates; $D = 1$

we observe that the simple LAB estimate no longer performs well. The other estimates, which take the structure of the tree into consideration in addition to using the node labels, perform better. In particular, note that the LB estimate (estimate 5) consistently outperforms all other estimates in this case. The only estimate that underperforms LAB is the LAB+SS estimate. In fact, a glance at the other figures in this series reveals that LAB+SS is almost always the worst performer. Intuitively, the reason for the poor performance of LAB+SS is that it places too much importance on structure. However, in cases when there are none or few differences between the input trees, this estimate performs well.

The results get more interesting as we increase the fraction of nodes with identical labels further. Consider Figure 9.14, which summarizes the result for $D = 0.4$. For the reasons stated above, LAB and LAB+SS remain the worst performers. The lower bound estimate LB continues to perform well. However, the two other estimates, PARM and LAB+POS, are also competitive. In fact, with very aggressive pruning, PARM outperforms LB. We believe this result is due to the fact that the PARM estimate can make more effective use of the fewer edges resulting from more aggressive pruning because of the following: The LB estimate can use missing edge information only when such missing edges are guaranteed to cause an increase in the edgewise cost (since it is a lower bound). In contrast, the PARM estimate can always use the missing edge information since it adjusts the likelihoods of various edit operations to account for the missing edges.

As the fraction of nodes with identical labels is increased further, we notice another interesting change. The results for $D = 0.6$ and $D = 0.8$, as summarized in Figures 9.15 and 9.16, are very similar to each other. In both cases, the estimation function 4, LAB+POS, is consistently the best performer by a significant margin. This result is explained by noting that at these high values of D, it is more prudent to match nodes based on their positions in the trees rather than by their labels. Further, recall from Figure 9.10 that when D is high, very few edges can be pruned even when we prune very aggressively. Thus estimates, such as LB and PARM, that rely on the absence of edges do not perform well. Finally, the simple estimates of LAB and LAB+SS perform poorly because of the diminished distinguishing ability of node labels.

Although Figures 9.15 and 9.16 are very similar, a careful observation reveals that as D is increased from 0.6 to 0.8, LAB+POS performs worse. Further, a glance at Figure 9.17 reveals that this trend continues when we increase D to 1. When the number of nodes with identical labels is relatively small, it is likely that these nodes are distinguishable using the structural properties used by LAB+POS. However, as this number increases, there is a higher likelihood of nodes with identical labels also having similar structural properties, thus reducing the effectiveness of LAB+POS in matching them properly.

9.3 Summary

In this chapter, we presented the results of our experimental evaluation of some tree differencing algorithms described in earlier chapters. We found that the technique we introduced in Chapter 5 for pruning edges from the induced graph is very successful. Conservative pruning results in a substantial reduction in the size of the induced graph for both real and synthetic data. As expected earlier, we found that aggressive pruning further reduces the size of the induced graph. Although aggressive pruning may in general result in deterioration of the quality of the final solution, we found that as we increase the value of the aggressiveness parameter A, the quality of the solution improves until we reach very high A values. The best choice for A depends on dataset, and it would be prudent to experiment with a few different values of A to determine a good choice. Based on our experiments, we believe that suitable values are likely to be in the range $[0.75, 0.95]$. For the eGuide dataset obtained from our implementation described in Chapter 8, we found that $A = 0.9$ produced good results. In general, we found pruning with high values of A to be very useful in both reducing the size of the induced graph and improving the quality of the solution.

Our study of edge cost estimation functions showed that, for our eGuide dataset, the estimate LB, based on the lower bound introduced in Section 5.5.2 of Chapter 5, almost always significantly outperforms the other estimates we studied. For our synthetic dataset, LB performs well when we do not have too many duplicate labels in the input data. When duplicate labels are very common, the estimate LAB+POS, which emphasizes the relative positions of nodes in their trees, outperforms LB. Again, given a new dataset, a good strategy would be to use some experimentation to determine a good edge cost estimation function. However, based on our experience with the C^3 system, we have found that the fraction of nodes with duplicate labels is typically small, suggesting that LB is a good default choice.

We also analyzed the running time of our implementation, and found it to be roughly proportional to the product of the sizes of the input trees. Our experiments indicated that our implementation spends a significant fraction of the running time on parsing the inputs, suggesting that we may be able to improve performance by using a more efficient parsing technique. A substantial fraction of the running time is also spent on evaluating the user-specified function used to compare labels. Our experiments used a function that compared labels using the difference between their character frequency histograms. In many applications, it may be possible to use a simpler and more efficient comparison function, thus improving the running time.

Chapter 10

Conclusion

In this dissertation, we motivated, formulated, and addressed the problem of managing evolving data that resides in a heterogeneous collection of autonomous databases. The principal contributions of this dissertation are summarized in Section 10.1 below. In Section 10.2, we discuss promising directions for future work in related areas.

10.1 Summary of Dissertation Results

First, we motivated the need for managing change in a heterogeneous collection of autonomous databases, and presented a framework for addressing this need. Next, we studied the problem of computing differences between snapshots of a database in this environment. We formulated a number of tree differencing problems, and presented algorithms to solve them. We then described the design and implementation of a database system for historical semistructured data. Finally, we described our implementation of a comprehensive change management system based on the techniques of this dissertation. We discuss these contributions in more detail below.

10.1.1 Change Management Framework

In Chapter 1, we introduced heterogeneous, autonomous databases, and described their growing importance. We motivated the need for database techniques that treat a collection of such databases as an integrated information system. In particular, we stressed the importance of techniques to manage the evolution of data in such an environment. We explained how the heterogeneity and autonomy of databases in this environment invalidate the assumptions made by conventional database techniques, and outlined the major research challenges in this area.

In Chapter 3, we presented our framework for managing change in heterogeneous, autonomous databases, and described how it builds on a framework of wrappers and mediators that is often used for data integration. An important design requirement, necessitated by the autonomy of the component databases, is that our system make very few assumptions about the component databases.

10.1.2 Differencing Algorithms

In Chapters 4, 5, and 6, we motivated the need for techniques for comparing two database snapshots (or partial snapshots) in order to detect changes. In addition to being the basis of an essential module in our change management system, such techniques are also useful in several other applications such as version control, syntactic program analysis, and automatic mark-up of changes in evolving documents (for example, manuals and legal documents). Data obtained from wrappers over heterogeneous databases has a hierarchical structure. We argued that differencing algorithms that are cognizant of such hierarchical structure produce results that are much more usable than those from algorithms that treat all data as strings or tables. We presented differencing algorithms for labeled trees, both ordered and unordered.

Two key features of our tree differencing techniques are the following: First, in addition to the node insertion and deletion and label update operations used by prior work, our algorithms also use expressive subtree operations such as move, copy, and uncopy. Using more expressive operations makes the problem of detecting changes harder, but produces results that are more usable. For example, if a paragraph in a manual is moved from one section to another, comparing the old and new manual using our techniques produces a corresponding subtree move operation as output, while earlier techniques that do not use such subtree operations produce a sequence of node insertion and deletion operations as output. Second, unlike prior work, we do not impose restrictions on the function used to compare node labels. Although such restrictions may seem reasonable at first glance, they have some serious ramifications that render techniques based on them unusable for our purposes. For example, most prior work requires that the function used to compare node labels be a distance metric. This function is required to satisfy an extended triangle inequality that asserts that the cost of updating one node label to another cannot be greater than the cost of deleting the first node and inserting the second. As a result of this restriction, such work does not allow us to specify, for example, that a node with label "movie" should not be matched to a node with label "restaurant" when we are comparing snapshots of the entertainment database described in Chapter 8. Our techniques, on the other hand, allow such specifications because of their flexible cost model.

In Chapter 4, we presented a technique for comparing ordered trees that achieves efficiency and optimality by using domain characteristics to simplify the problem. In Chapter 5, we presented a more general solution for unordered trees, and in Chapter 6, we described a model of tree transformations that is more declarative than the traditional edit script model, and that leads to simpler algorithms for solving the differencing problem. In Chapter 8, we described how we use our tree differencing algorithms to detect changes in semistructured data in the OEM model by mapping tree edit operations to edit operations in OEM.

10.1.3 Database System for Historical Semistructured Data

As described in Chapter 3, semistructured data is data that has structure that may be irregular, incomplete, and dynamic. We motivated the need for data management techniques for semistructured data, and explained why traditional database techniques cannot be directly applied to such data. We focused on difficulties encountered in modeling historical semistructured data; that is, semistructured data together with its history of changes. In Chapter 7, we presented a data model, DOEM, and a query language, Chorel, for historical semistructured data. A key feature of DOEM and Chorel is the use of an explicit representation of changes as first-class entities instead of an implicit representation of changes as the difference between two database states.

We described the implementation of CORE, a database system for historical semistructured data. Our implementation strategy avoids reimplementation of several database modules by using the existing implementation of Lore, a database system for semistructured data, and Lorel, Lore's query language. In addition to being modular, this strategy also permits us to implement CORE by using other database systems, such as the O_2 object database system [BDK92]. We represent DOEM data, which is logically an annotated graph, in OEM, which is an ordinary graph model, by encoding annotations using special objects. Chorel queries over a DOEM database are implemented by translating them into equivalent Lorel queries over the OEM encoding of the DOEM database.

10.1.4 The C^3 System

The work described in this dissertation has been implemented as the C^3 system for managing change in heterogeneous, autonomous databases. The C^3 system uses our differencing algorithms to detect changes in heterogeneous, autonomous databases (called *source databases*), and our implementation of CORE to store and query the history of these changes along with the base data. We have also implemented a subscription service that notifies subscribers of changes of interest to them. Such subscriptions are specified using a special form of Chorel queries, and are extremely powerful. The C^3 system builds on companion work in data integration and semistructured data. We use an architecture of wrappers and mediators, with wrapper and mediator generation techniques from the Tsimmis project. As described above, we also use the Lore database system for semistructured data. The combination of the techniques in this dissertation with those from the Tsimmis and Lore projects results in an extremely versatile system for managing change in heterogeneous, semistructured databases. In Chapter 8, we described our experiences with the C^3 system. We described the functionality offered by the system to discover and study the evolution of data in heterogeneous, semistructured databases. We also described how the system modules, based on work described earlier, cooperate to support this functionality.

10.2 Future Work

In this section, we describe some opportunities for future work in topics related to this dissertation. We classify such opportunities into three categories: comparing data, managing historical semistructured data, and extending the C^3 system.

10.2.1 Comparing Data

In Chapters 4–6, we noted the advantages of describing differences between trees using not only node insertion, deletion, and label update operations, but also the subtree operations of move, copy, and uncopy. In essence, using more expressive operations results in edit scripts that are more meaningful than edit scripts that use only simple node operations. It would be interesting to study whether we can push this strategy further; that is, would using even more expressive operations lead to still better results? For example, consider the comparison of structured documents, modeled as ordered trees. We may wish to use a *merge* operation that combines two or more sibling nodes into one node by concatenating their contents. Thus, a merge operation could be used to combine the sentences from three paragraphs into one paragraph or to combine the contents of two sections into one.

It seems reasonable to assume that such additional edit operations would result in more usable descriptions of changes. For example, an edit script that indicates that three paragraphs in a document were merged is more succinct and intuitively more convenient than one that indicates that two paragraphs were deleted, with their constituent sentences moved to the third. However, such a proliferation of edit operations raises two important issues that need to be resolved: First, without any additional restrictions, such new edit operations may interfere with one another in unexpected ways, leading to unintuitive edit scripts similar to those described in Chapter 5. Second, it is not clear how our general strategy of mapping edit scripts to compact representations (matchings or signatures) would generalize to include such complex edit operations.

We could carry the idea of more complex edit operations even further by allowing user-defined edit operations. That is, the problem inputs consist of not only the two trees to be compared, but also a specification of the edit operations with which the differences are to be described. A general solution to this problem is likely to be intractable. However, the problem can be simplified by imposing domain-based or domain-independent restrictions, and by relaxing the requirement that the solution be optimal.

It is natural to consider the extension of our work on comparing trees to techniques for comparing more general graph structures. In Chapter 8, we described how our techniques are applied to graph structured data that has a preferred spanning tree. However, when the data is truly graph structured, we need to devise more general graph differencing techniques. As we have done for trees, the first step is deciding on the set of edit operations on graphs. The simple edit operations of node insertion and deletion, and arc addition and removal, are obvious candidates. However, just as we obtain better results for trees by including subtree operations, we may obtain better results for graphs by including more complex graph edit operations. For example, we may define a merge operation that replaces two nodes in a graph with a new node with a label that is a concatenation of the labels of the original nodes, with arcs incident on the original node redirected to the new node. An efficient algorithm for an optimal solution of a general formulation of this problem is unlikely. However, by imposing suitable restrictions and carefully designing a mapping between edit scripts and signatures, we could use a strategy analogous to our strategy for trees. Further, we could relax the requirement that the solution be optimal, and use heuristic descriptions of good solutions.

Another avenue for future work is devising algorithms for comparing data that is too large to fit in primary storage and must therefore be accessed off secondary or tertiary storage such as magnetic and optical disks, magnetic tapes, and juke-boxes. For example, we may wish to compute differences between two versions of an engineering design, a large manual, or the hierarchy of documents on a Web site. Accesses to secondary storage are typically several orders of magnitude slower than accesses to primary storage. Further, secondary storage accesses are typically faster if data is accessed sequentially or in a clustered manner. Since the algorithms described in earlier chapters do not take these factors into account, a naive implementation of them for secondary storage is likely to be impractical. A simple method to compute differences between datasets in secondary storage is to divide each dataset into fragments that fit in main memory, compute the differences between pairs of these fragments using main memory algorithms, and combine the differences thus detected. However, such a strategy is likely to detect a large number of spurious differences due to the possibility of mismatched fragments. It may be possible to partially amend this situation by devising heuristics that reduce the likelihood of mismatched fragments, and by postprocessing the detected differences to make local improvements.

10.2.2 Managing Historical Semistructured Data

Recall, from Chapter 7, that CORE uses an implementation strategy that is based on encoding DOEM in OEM and translating Chorel queries to equivalent Lorel queries over the OEM encoding. The disadvantage of this strategy is that query processing is often very inefficient. The Lorel queries produced by our translation scheme often involve several joins and nested quantifications. We may be able to ameliorate many of these performance problems by modifying Lore to generate better query plans for the kinds of queries produced by CORE. We can index strategic data, such as annotation sets, using conventional indexing techniques, and modify the Lore query optimizer to use these indexes. Such a solution should be easy to incorporate into our current implementation.

As an enhancement to the above solution, we can design and implement indexes that are specialized for historical data. For example, such indexes can be biased to account for the fact that recent data and changes are more likely to be accessed than those in the more distant past. Instead of treating all indexed data as equally important, biased indexes prioritize access to recent data. For example, such an index would allow the retrieval of annotations that were added this week to be much faster than the retrieval of annotations that were added a year ago. The design of such *biased* indexes, along with algorithms to build and incrementally maintain them as data evolves, is a promising topic for future work.

In addition to the traditional value-based indexes, semistructured databases can also benefit from *path indexes* that index an object based on the values of its subobjects nested several levels deep. While such path indexes can speed up query processing, they are often expensive to maintain in the face of frequent updates to the database. Thus, in addition to designing efficient implementations of path indexes, we need to devise techniques to determine which indexes are most beneficial for a given query and update mix [CCY94].

Another approach to improving the performance of Chorel queries is to implement CORE directly instead of using the encoding and translation scheme. In addition to avoiding the complicated queries that arise as artifacts of our encoding and translation scheme, this approach would permit the low-level design of the database system to be optimized for DOEM and Chorel.

In DOEM, annotations describing the history of changes to a node or arc are conceptually attached to that node or arc. We have seen that this model permits intuitive browsing and querying of historical data. However, an implementation of DOEM is not required to physically colocate annotations with the nodes or arcs they are conceptually attached to. For example, it may be more efficient to store all annotations separately, perhaps organized using a biased index as described above. An interesting general problem in this area is the following: Given a large labeled, directed graph (optionally with annotations on the nodes and arcs), and some description of likely access patterns (for both retrieval and modification), what is the most efficient way to represent the graph on disk?

As described in Chapter 7, a central construct in Chorel is an annotated path expression. Annotated path expressions are simply path expressions whose components may be modified by an optional annotation expression. For example, the annotated path expression `a.<add at T>b` denotes a path consisting of an a-edge followed by a b-edge that has an $add(t)$ annotation, with t bound to T. In Chapter 7, we focused on simple annotated path expressions in which annotation expressions modify only simple path expression components, that is, single labels. General path expressions may contain components that use wildcards and regular expression operators, and the ability to modify such components using annotation expressions is often useful. For example, we may use the annotated path expression `a.?<add>.c` to denote a path consisting of an *a*-edge followed by an edge (with any label) that has an *add* annotation, followed by a *c*-edge. As another example, consider the expression `a.(b<add>|c<rem>).#<add>`. Intuitively, this annotated path expression suggests a path consisting of an *a*-edge followed by either a *b*-edge with an *add* annotation or a *c*-edge with a *rem* annotation, followed by zero or more edges with *add* annotations. This interpretation implicitly assumes that the `<add>` annotation expression attached to the closure (`#`) operator denotes the presence of an *add* annotation on every edge included in the closure. An alternate interpretation is to only require an *add* annotation on the last edge included in the closure. Yet another interpretation is to require an *add* annotation on some edge in the closure. The last two interpretations need to handle the special case of the closure including no edges. In general, it should be interesting to explore the options for attaching annotation expressions to path expressions in a more flexible manner. For example, we may wish to use `a.(b(<add>|<rem>)|c(<add>&<rem>))` to denote a path consisting of an *a*-edge followed by either a *b*-edge with an *add* or a *rem* annotation, or by a *c*-edge with both an *add* and a *rem* annotation.

Recall from Chapter 7 that virtual annotations provide convenient access to information that is implicitly represented in a DOEM database. We have seen some examples of such annotations: *at*, *snap*, *during*, and *ov* (old value). However, these virtual annotations are hard-coded as part of the CORE implementation. We cannot introduce new virtual annotations without modifying our implementation. Given our translation-based implementation scheme, such modifications are not difficult to make. However, it would be interesting to design and implement a facility that allows new kinds of virtual annotations to be defined and used at the CORE user interface, without any modifications to the CORE implementation. Such a facility would provide functionality somewhat similar to that provided by views in a traditional database system.

In addition to extending querying facilities for historical semistructured databases as described above, it should be interesting to implement a trigger facility for such databases. Conventional database systems often include trigger facilities that permit a database system to automatically respond to the occurrence of certain kinds of events [WC96a]. Triggers are commonly expressed using an event-condition-action (ECA) construct that specifies the action to be performed when events of a certain class occur, and when the specified condition holds true. It should be interesting to extend Lore to include such a trigger facility for semistructured data. The DOEM update model is a good basis for the event language of such triggers. For example, we may associate a trigger with an event that adds an edge with a specified label. The condition and action action of such triggers can be specified using standard Lorel query and update statements. It would be interesting to study the tradeoff between generality and implementation efficiency in such a trigger facility.

10.2.3 Extending the C^3 System

Recall from Chapter 8 that the C^3 system allows us to integrate heterogeneous, autonomous databases, to detect changes in these databases, to store an integrated historical database describing the data and changes of interest to us, to query this historical database using a general-purpose query language, and to request notification of interesting changes specified using a powerful subscription language. Together, these facilities provide a comprehensive system to monitor and study the evolution of data in the source databases. A logical next step in change management for heterogeneous, autonomous databases involves adding the ability to effect change at the source databases. Given the autonomy of the source databases, a strategy that requires permissions to directly modify the source databases is likely to be unsuccessful in practice. In many cases, the source databases may offer external agents (such as the wrappers used by C^3) no facilities for modifying the information they contain. In other cases, a source database may offer some rudimentary and restrictive mechanisms for modifying the data. For example, many databases on the Web, such as the Internet Movie Database, offer a forms interface that external agents can use to suggest changes to the database [IMD98]. Such interfaces need to be modeled carefully in order to accurately capture their semantics, which are often more complex than the simple atomic update semantics used in traditional database systems. For example, the Internet Movie Database offers forms for several purposes, such as voting on the quality of a movie, adding missing information, correcting existing information, and submitting a review. Successful submission of a form does not guarantee that the suggested modification has occurred, or even that it will occur in the future. Further, different types of suggested modifications have different chances of actually being made. For example, a simple vote rating a movie as good is likely to be accepted automatically; however, a contentious claim regarding the true origin of a script is likely to be rejected or revised by a human being reviewing such claims. A successful strategy for implementing a facility for effecting changes in source databases needs to reflect some of these intricacies of the update interfaces offered by the source databases. In general, an update facility for C^3 requires modeling of long-running activities consisting of several steps, some of which involve human interaction. In this respect, such a facility is related to prior work on *long-running transactions* and *workflows* and we may be able to use some techniques from those fields [GMS87, WWW$^+$97]. A fully general framework for effecting changes in autonomous databases may be very ambitious because such a framework would require solving, among other problems, a particularly troublesome variant of the view update problem: The problem of mapping changes specified in the integrated OEM view of the source data to the operations supported by the source's modification interface is likely to be intractable. Fortunately, we may be able to obtain significant benefits by implementing a restricted modification framework based on simple ideas. For example, it is not difficult to see how the forms-based modification interface supported by the Internet Movie Database could be mapped to changes in the OEM view of the database. A very simple extension to the C^3 system would allow users to make only those changes to the OEM view that can be unambiguously and easily mapped to the forms supported by the source database. This approach is similar to that used to define updateable views in SQL [DD93].

A module to monitor and maintain inter-database integrity constraints would be another useful extension to the C^3 system. For example, as described in Chapter 1, at Stanford there are a several databases that store personnel information for people in the Computer Science department. There are several integrity constraints spanning these databases, a simple one being that the primary phone number listed for a person be the same in all databases. Suppose we have integrated these databases using the C^3 system. Using QSS, it is easy to create a subscription that notifies a person whenever that person's phone number is listed inconsistently by these databases for longer than, say, three days. On receiving such a notification, this person may then take the actions necessary to correct the situation. In many cases, instead of only notifying users of inconsistencies, it may be possible to suggest one or more mechanisms to resolve the inconsistency. In our example above, the system could suggest that the phone number from the most recently updated database be propagated to the others. As discussed earlier, such actions may be represented using a workflow that includes, in addition to actions updating the source databases, actions requiring user approval. In order to implement such a strategy, we need methods for automatically or semi-automatically generating consistency restoring workflows from a declarative specification of inter-database integrity constraints. There is a substantial body of work in the related area of integrity constraint management for conventional database systems [WC96b]. In [CGMW94, CGMW96], we describe a simple rule-based framework and toolkit for constraint management in autonomous databases. Extending such work to semistructured data in an autonomous environment is a fruitful topic for future research.

Recall, from Chapter 7, that our implementation of a query subscription service (QSS) maintains a DOEM database for each subscription. New changes periodically detected by C^3 are added to this DOEM database. Over time, the DOEM database of a subscription potentially grows without bound as changes accumulate. In practice, we need some method to bound the size of these databases. (In our current implementation, we simply suspend servicing subscriptions whose DOEM databases grow beyond a fixed limit.) For some subscriptions, it may be impossible to accurately service the subscription without storing an unbounded amount of historical information in its DOEM database. For example, consider a subscription that asks for all the times at which an object was modified since a fixed date; servicing this subscription requires that we store the entire history of modifications to the specified object. However, for many subscriptions, we do not need to store the entire history in this manner. As a simple example, consider a subscription that asks only for newly added objects; for this subscription, we need to store only the most recent polling query result in the DOEM database. In addition, it may often be possible to service a subscription by storing only a fraction of its complete DOEM database as defined in Chapter 7. These observations suggest a *DOEM pruning* problem: Given a subscription, determine the least amount of information that must be stored in the subscription's DOEM database in order to correctly service the subscription. Note that we need an online solution to the DOEM pruning problem. That is, every time we receive new changes, we need to determine which changes need not be installed, and which old changes and data may be discarded without affecting current and future subscription results. A precise solution to this problem is likely to be complex; however, several approximate or heuristic strategies may yield satisfactory results. For example, it may be possible to use a collection of simple rules that indicate what to prune. One such rule may indicate that if a subscription's filter query mentions only *add* annotations, then *rem* annotations need not be stored unless they are the most recent annotations on their respective arcs. Studying such strategies for DOEM pruning, and their effects on the performance and accuracy of QSS is an interesting topic for future work.

Another method for saving space in the QSS implementation is the sharing of DOEM databases among subscriptions. Our current implementation maintains a separate DOEM database for each subscription. However, given several similar subscriptions, it may be advantageous to combine their DOEM databases in order to save storage space and improve performance. As a very simple example, if two or more subscriptions have identical polling queries and polling frequencies, we can use a single DOEM database to service them both. Note that our menu-driven interface to QSS makes it likely that several users chose the same polling query. Further, it may often be possible to service a subscription approximately by using the DOEM database of another subscription. Exploring such opportunities for *DOEM sharing* among related subscriptions is an interesting topic for future work.

The C^3 system detects changes by polling the source databases, which are accessed using wrappers that present a simple query interface. However, in some cases source databases may offer notification facilities. For example, an online retailer may offer to notify us when certain books are available, or when the price of a computer monitor drops below a specified threshold [AMA98, CDW98]. If such facilities exist, ignoring them and using only polling and differencing to detect changes is wasteful. It would be useful to extend C^3 to include *active wrappers* that map such notification services to DOEM histories. For example, an active wrapper for the online retailer mentioned above would map an email message indicating the availability of a book to a set of change operations on the OEM representation of the source data. We can adapt many of the template-based wrapper implementation techniques used for Tsimmis wrappers to such active wrappers. However, unlike regular wrappers, active wrappers need a per-subscription set-up. That is, in addition to translating notifications from the source model to OEM, active wrappers need to first indicate to the source the kinds of notifications they wish to receive by creating a source subscription. The subscription services offered by source databases vary considerably; thus we need methods to integrate not only the data models and query languages of source databases, but also their subscription services. The design of methods for such integration of subscription services is an interesting topic for future research. An implementation of such methods in a toolkit for rapid construction of active wrappers would make a valuable addition to the C^3 system.

There are several opportunities for future work on user interfaces to the C^3 system. An interface to semistructured databases (both historical and non-historical) would benefit from a facility that permits the objects in a query result to be not only browsed, but also marked and selectively used in subsequent queries. For example, suppose we issue a query to find authors who have published a paper whose title contains the word historical. The user interface may display, say, fifty authors that qualify. Next, we may browse the details for these authors and, based on our browsing, mark some of the authors as interesting. We may now wish to find books written by one of these interesting authors on a given topic. The design and implementation of a user interface that supports such closely coupled querying and browsing presents several interesting challenges. For example, given that the query-browse-mark-query cycle may be repeated several times, we need a method to efficiently evaluate a composite query containing some combination of past queries and marked objects from past query results.

Bibliography

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[ACHK93] Y. Arens, C. Chee, C. Hsu, and C. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, June 1993.

[ACM95] S. Abiteboul, S. Cluet, and T. Milo. A database interface for file update. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[ADD+94] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M.-C. Shan. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1994.

[AK97] N. Ashsish and C. Knoblock. Wrapper generation for semi-structured internet sources. In *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, 1997.

[AMA98] The amazon.com online bookstore. `http://www.amazon.com/`, 1998.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, November 1996.

[Arm74] W. Armstrong. Dependency structures of database relationships. In *Proceedings of the IFIP Conference*, pages 580–583, 1974.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montréal, Québec, June 1996.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis:. *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann, 1992.

[BKKK87] J. Banerjee, W. Kim, H. Kim., and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311–322, 1987.

[BLCG92] T. Berners-Lee, R. Cailliau, and J. Groff. The world-wide web. *Computer Networks and ISDN Systems*, 25:454–459, 1992.

[BLT86] J. Blakeley, P.-A. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., June 1986.

[BN98] The Barnes and Noble online bookstore.
`http://www.barnesandnoble.com`, 1998.

[BPSM98] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. World Wide Web Consortium Recommendation. Available at `http://www.w3.org/TR/1998/REC-xml-19980210`, February 1998.

[Buc96] A. Buchmann. The active database management system manifesto: A rulebase of ADBMS features. *ACM SIGMOD Record*, 25(3):20–49, September 1996.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[Cat96] R. Cattell. *The Object Database Standard: ODMG-93 Release 1.2*. Morgan Kaufmann Publishers, San Francisco, California, 1996.

[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.

[CAW99] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying history and changes in semistructured data. *Theory and Practice of Object Systems*, 1999. To appear. Available at `http://www-db.stanford.edu`.

[CCY94] S. Chawathe, M-S. Chen, and P. Yu. On index selection schemes for nested object hierarchies. In *Proceedings of the International Conference on Very Large Data Bases*, pages 331–341, 1994.

[CDN98] The cdnow.com online music store. `http://www.cdnow.com`, 1998.

[CDW98] The CDW online computer store. `http://www.cdw.com/`, 1998.

[CGL+97] S. Chawathe, V. Gossain, X. Liu, J Widom, and S. Abiteboul. Representing and querying changes in heterogeneous semistructured databases (demonstration description). Technical report, Stanford University Database Group, November 1997. Available at `http://www-db.stanford.edu`.

[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.

[CGMH$^+$94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis project: Integration of heterogeneous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, October 1994.

[CGMW94] S. Chawathe, H. Garcia-Molina, and J. Widom. Constraint management for autonomous distributed databases. *Data Engineering Bulletin*, 17(2):23–27, 1994.

[CGMW96] S. Chawathe, H. Garcia-Molina, and J. Widom. A toolkit for constraint management in heterogeneous information systems. In *Proceedings of the International Conference on Data Engineering*, pages 56–65, 1996.

[CHS$^+$95] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, and R. Fagin. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, pages 123–130, Los Angeles, California, 1995.

[Clu98] S. Cluet. Designing OQL: allowing objects to be queried. *Information Systems*, 23(5):279–305, July 1998.

[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.

[DD93] C. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Massachusetts, 1993.

[DHR96] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montréal, Québec, 1996.

[EG98] The Gate eGuide. `http://www.sfgate.com/eguide/`, 1998.

[FGM$^+$97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol—HTTP/1.1. Available at `http://www.w3.org/Protocols/rfc2068/rfc2068`, January 1997. Network Working Group Request for Comments 2038.

[GCCM96] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A standard textual interchange format for the Object Exchange Model (OEM). Technical report, Stanford University Database Group, 1996. Available at `http://www-db.stanford.edu/`.

[GH97] T. Griffin and R. Hull. A framework for implementing hypothetical queries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 231–242, Tucson, Arizona, May 1997.

[GHJ$^+$93] S. Ghandeharizadeh, N. Hull, T.D. Jacobs, J. Castillo, M. Escobar-Molano, , S.-H. Lu, J. Luo, C. Tsang, and G. Zhou. On implementing a language for specifying active database execution models. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, September 1996.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, California, December 1987.

[Gol90] C. Goldfarb. *The SGML handbook.* Oxford University Press, 1990.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of* *the Twenty-third International Conference on Very Large Data Bases*, Athens, Greece, 1997.

[HBGM+97] J. Hammer, B. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, and R. Yerneni. Template-based wrappers in the Tsimmis system. In *Proceedings of the Twenty-Third ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 18–25, Tuscon, Arizona, May 1997. Available at `http://www-db.stanford.edu`.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[HHS+98] M. Haertel, D. Hayes, R. Stallman, L. Tower, P. Eggert., and W. Davison. The GNU diff program. Texinfo system documentation, 1998. Available through anonymous FTP at `prep.ai.mit.edu`.

[HSF85] K. Harrenstien, M. Stahl, and E. Feinler. Nicname/Whois. Technical report, SRI International, October 1985. Internet Engineering Task Force Network Working Group RFC 954.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, Montreal, Canada, June 1996.

[IMD98] The Internet Movie Database. `http://www.imdb.com/`, 1998.

[Inm92] W. Inmon. EIS and the data warehouse: A simple approach to building an effective foundation for eis. *Database Programming and Design*, 5(11):70–73, November 1992.

[JUN98] The Junglee online shopping guide. Available at `http://www.junglee.com/wcomm/wcoverview.html`, 1998.

[Kif95] M. Kifer. EDIFF—A comprehensive interface to diff for Emacs 19. Available through anonymous FTP at `ftp.cs.sunysb.edu` in `/pub/TechReports/kifer/ediff.tar.Z`, 1995.

[KL86] B. Kantor and P. Lapsley. Network news transfer protocol. Technical report, University of California, San Diego, February 1986. Internet Engineering Task Force Network Working Group RFC 977.

[KLSS95] T. Kirk, A. Levy, J. Sagiv, and D. Srivastava. The information manifold. Technical report, AT&T Bell Laboratories, 1995.

[Knu86] D. Knuth. *Computers and Typesetting.* Addison-Wesley, Reading, Massachusetts, 1986.

[KRO98] Online traffic updates from KRON Newscenter 4. `http://www.sfgate.com/traffic/`, 1998.

[Lam94] L. Lamport. *Latex: A Documentation Preparation System User's Guide and Reference Manual.* Addison Wesley Longman, Inc., July 1994.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[LND98] The Lands' End online retail store. `http://www.landsend.com`, 1998.

[LYV$^+$98] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability based mediation in Tsimmis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 564, Seattle, Washington, June 1998.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MBL98] The musicblvd.com online music store. `http://www.musicblvd.com`, 1998.

[Mel96] J. Melton. An SQL3 snapshot. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 666–672, New Orleans, Louisiana, February 1996.

[MW98] J. McHugh and J. Widom. Query optimization for semistructured data. Technical report, Stanford University Database Group, 1998. Available at `http://www-db.stanford.edu/`.

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

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of the International Conference on Data Engineering*, pages 79–90, 1997.

[NYT98] The New York Times online. `http://www.nyt.com`, 1998.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Data Bases*, pages 413–424, Bombay, India, September 1996.

[PAW98] The Palo Alto Weekly online, 1998. `http://www.service.com/PAW/`.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 161–186, Singapore, December 1995.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A mediation system based on declarative specifications. In *Proceedings* *of the International Conference on Data Engineering*, pages 132–141, New Orleans, February 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[Pos82] J. Postel. Simple mail transfer protocol. Technical report, Information Sciences Institute, University of Southern California, Marina del Rey, California, August 1982. Internet Engineering Task Force Network Working Group RFC 821.

[PR85] J. Postel and J. Reynolds. File transfer protocol (FTP). Technical report, Information Sciences Institute, University of Southern California, Marina del Rey, California, October 1985. Internet Engineering Task Force Network Working Group RFC 959.

[PS82] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice-Hall, 1982.

[QWG+96] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A Lightweight Object REpository for semistructured data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[RHe98] D. Raggett, A. Le Hors, and I. Jacobs (eds.). HTML 4.0 specification. Available at `http://www.w3.org/TR/REC-html40/`, April 1998.

[Rot] E. Rothberg. The *wmatch* program for finding a maximum-weight matching for undirected graphs. Live OR collection. Available at `http://www.orsoc.org.uk/home.html`.

[SA86] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

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

[SL90] A. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SLS$^+$93] K. Shoens, A Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The rufus system: Information organization for semistructured data. In *Proceedings of the International Conference on Very Large Data Bases*, pages 97–107, Dublin, Ireland, August 1993.

[Soo91] M. Soo. Bibliography on temporal databases. *SIGMOD Record*, 20(1):14–24, March 1991.

[SWZS94] D. Shasha, J. Wang, K. Zhang, and F. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):668–678, April 1994.

[SZ90] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11:581–621, 1990.

[Ukk85] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Uni93] International Telecommunication Union. Specification of Abstract Syntax Notation One (ASN.1). Technical report, Telecommunication Standardization Sector of ITU, 1993. ITU-T Recommendation X.208. Available at `http://www.itu.int/`.

[UW97] J. D. Ullman and J. Widom. *A first course in database systems.* Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[Vix98] P. Vixie. Red Hat Linux system manual for cron. Available at `http://www.redhat.com`, 1998.

[W3C98] The World-Wide Web Consortium online. `http://www.w3.org/`, 1998.

[Wag75] R. Wagner. On the complexity of the extended string-to-string correction problem. In *Seventh ACM Symposium on the Theory of Computation*, 1975.

[WC96a] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann Publishers, San Francisco, California, 1996.

[WC96b] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1996.

[WCS96] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl.* O'Reilly, second edition, 1996.

[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.

[Wid96] J. Widom. Integrating heterogeneous databases: Lazy or eager? *ACM Computing Surveys*, 28A(4), December 1996.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

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

[WP98] The Washington Post online. `http://www.washingtonpost.com`, 1998.

[WWW+97] D. Wodtke, J. Weissenfels, G. Weikum, A. Dittrich, and P. Muth. he mentor workbench for enterprise-wide workflow management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, June 1997.

[WZC95] J. Wang, K. Zhang, and G. Chirn. Algorithms for approximate graph matching. *Information Sciences*, 82:45–74, 1995.

[WZS95] T-L. Wang, K. Zhang, and D. Shasha. Pattern matching and pattern discovery in scientific, program, and document databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1995.

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

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[Zha95] K. Zhang. Personal communication, May 1995.

[Zim90] D. Zimmerman. The finger user information protocol. Technical report, Center for Discrete Mathematics and Theoretical Computer Science, December 1990. Internet Engineering Task Force Network Working Group RFC 1196.

[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.