SEARCHING ALGORITHMS AND DATA STRUCTURES FOR COMBINATORIAL, TEMPORAL AND PROBABILISTIC DATABASES

By

Rosalba Giugno

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
"DOTTORE DI RICERCA"
AT
UNIVERSITÀ DEGLI STUDI DI CATANIA
CATANIA, ITALY
DECEMBER 10, 2002

UNIVERSITÀ DEGLI STUDI DI CATANIA

Facoltà di Scienze Matematiche Fisiche e Naturali

Dottorato di Ricerca in Informatica

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Searching Algorithms and Data Structures for Combinatorial, Temporal and Probabilistic Databases**" by **Rosalba Giugno** in partial fulfillment of the requirements for the degree of **"Dottore di Ricerca"**.

Dated: December 10, 2002

Tutor: Prof. Alfredo Ferro, Prof. Dennis Shasha

Coordinator: Prof. Alfredo Ferro

UNIVERSITÀ DEGLI STUDI DI CATANIA

Date: **December 10, 2002**

Author: **Rosalba Giugno**

Title: **Searching Algorithms and Data Structures for Combinatorial, Temporal and Probabilistic Databases**

Department: **Mathematics and Computer Science**

Degree: **"Dottore di Ricerca"**

Permission is herewith granted to Università degli Studi di Catania to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

To "The Ocean"

Table of Contents

Abstract

Next-generation database systems, dealing with XML, Web, network directories and structured documents, often model the data as trees and graphs. In addition, new database systems arising in such diverse fields like molecular databases, image databases, or commercial databases also use a graph or tree model to represent data. Due to the importance of graph and tree databases, several algorithms have been developed for tree and graph querying.

In the first part of this thesis, we study methods for searching in graphs and trees. We propose a novel application-independent method for finding all the occurrences of a subgraph in a database of graphs. Our method enumerates and hashes node-paths of the database graphs in order to create a fingerprint of the database. Given a query, we also build its fingerprint, and we use it to filter the database. We have implemented the proposed method in a software package called GraphGrep. Our software has been tested on random databases and molecule databases. We have compared GraphGrep with the best known software, (Daylight and Frowns) and we have obtain very promising results. In addition, we propose an optimization method for indexing techniques which are used for fast retrieval of subtrees from a database of trees.

Besides the need for searching and querying trees and graphs there are numerous applications that require temporal uncertainty associated with database objects. Examples are commercial package delivery databases, financial databases and weather databases. Although there has been extensive work on temporal databases and temporal object oriented databases, there has been less work on probabilistic object bases and to our knowledge no work on temporal probabilistic data in object bases.

In the second part of the thesis, we discuss temporal probabilistic object bases. To automatically manipulate time and uncertainty information we propose a data model and an algebra for temporal probabilistic object data bases (TPOBs). This allows us to associate events with sets of time points and probabilities. We distinguish between explicit object

base instances, where the sets of time points along with their probability intervals are enumerated, and implicit ones, where the sets of time points are expressed by constraints and their probability intervals by probability distribution functions. We extend the relational algebra to both explicit and implicit instances and prove the operations on implicit instances correctly implement their counterpart on explicit instances.

Acknowledgements

The four years of my Ph.D have been a very rich experience of my life. I was fortunate to work with several brilliant people in great places: University of Catania, University of Maryland, University of Rome and New York University. They embraced me with their friendship and devoted time and energy to improve my academic skills; their different personalities helped me improve myself in my work as well in my way to be. Leaving abroad gave me the possibility to get acquainted with different cultures, to encounter different problems, to visit beautiful places and to meet close friends.

I am deeply grateful to my advisor Alfredo Ferro from University of Catania for his great optimism, for his continuous encouragement, his constant support in everything I have worked on during these years, and for his fathering affection and teachings. I am also deeply grateful to my co-advisor Dennis Shasha from New York University for his friendliness, his great hospitality, and for all his advice and help.

I am thankful to Giovanni Gallo from University of Catania for friendly and patiently introducing me to academic research; to VS Subrahmanian from University of Maryland for his teaching and his support; to Angelo Gilio from University of Rome for his friendship and for guiding me in the world of the probability; to Davi Geiger from New York University with whom I really enjoyed to work; to Thomas Lukasviewicz and Chih-Yi Hsu for all their work on our projects.

I owe so much to my family and to my close friends for all the support and love they have given to me. Their love makes possible everything I do.

Catania, Italy — Rosalba Giugno
December 10, 2002

Chapter 1

Introduction

In the last decades relation databases have become a powerful technology for data manipulation. In the recent years however, rapid developments in science and technology, had an profound effect on the way data is represented. A new problem has emerged in the database technology: *new data forms cannot represented efficiently with classical tables*. Many applications represent data as graphs and trees; other applications require (from a database system) the ability to manipulate time and uncertainty. For this reason, several research efforts have been initiated in an attempt to formulate models for the *"next generation databases"*: databases able to represent novel types of data, provide novel manipulation capabilities while efficiently supporting standard searching operations.

1.1 Searching in Graphs

Graphs are used in many applications to represent data due to their general, flexible and powerful structure. A graph is composed of a set of vertices and a set of edges (links) between pairs of vertices. Usually, vertices represent data (i.e. anything we want to describe) and edges represent the relations among them. Depending of the level of abstraction used to represent the data, a graph provides either a syntactic or semantic description of the data (see for example Fig. 1.2).

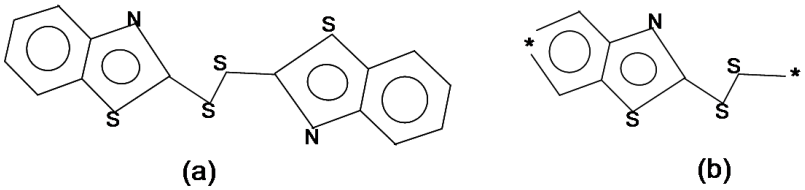


Figure 1.1: (a) A Chemistry compound. (b) A query containing wildcards. Their structures are naturally describe by graphs.

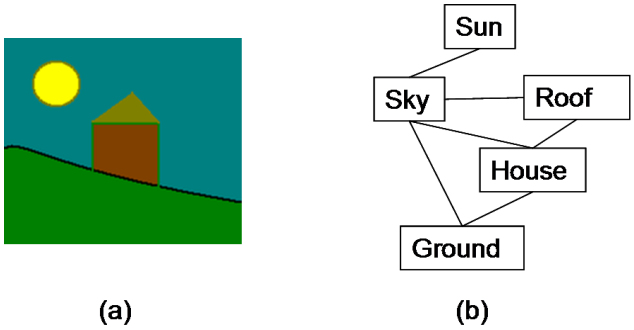


Figure 1.2: (a)An image and (b) a RAG (Region Adjacent Graph) representation of the image.

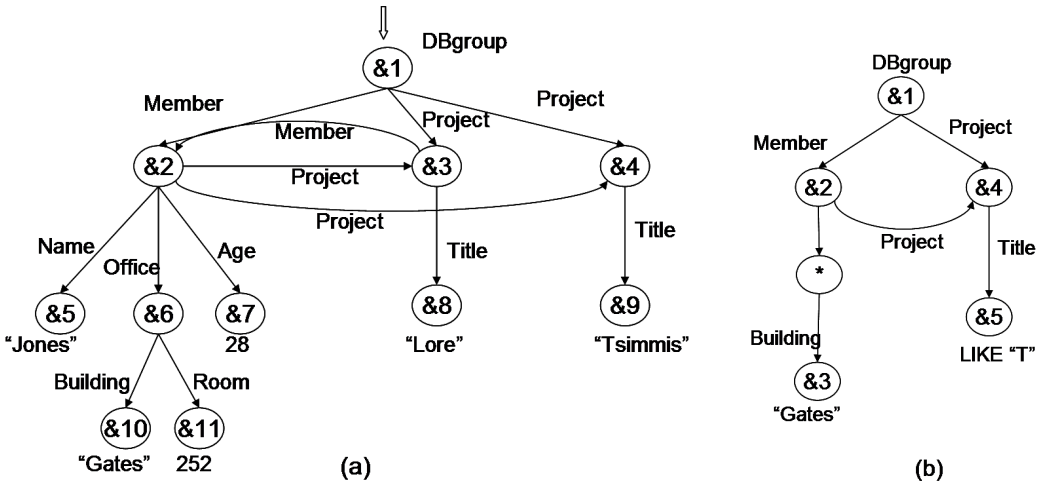


Figure 1.3: (a) A structured database. (b) A query containing wildcards.

- We first consider a biochemical database, a database of proteins [97, 18, 146]. Proteins are naturally represented as labelled graphs (Fig. 1.1 (a)): the vertices represent the atoms and the edges the links between the atoms. The proteins in a database are usually classified based on their common structural properties. One application of such classification is the prediction of the functionality of a new protein fragment (discovered or synthesized). This functionality can be deduced by locating the common structures between the new fragment and known proteins. Finally, the queries may contain wildcards which match a vertex or a path in the data (Fig. 1.1 (b)).

- Visual languages have graphs as the underlying data model. These languages are used in software engineering to specify projects and tools for integrated environments [70], in CIM (Computer Integrated Manufacturing) systems describing process modelling [153] and in visual databases for defining the semantics of the query languages [11].

- In computer vision graphs can be used to represent images in different levels of abstraction [159, 179]. In a low-level representation the vertices of a graph correspond to pixels and the edges to spatial relations between pixels [144]. In higher levels of description, an image is represented by a graph (RAG) as follows (Fig. 1.2) : the image is decomposed in regions, the regions are the vertices of the graph and the edges are the spatial relations between the regions [95, 121].

- The Web, network directories and semi-structured databases often model the data as graphs [3, 74, 59, 86] (Fig. 1.3 (a)). Usually, such databases are large directed labelled graphs where vertices are atomic or complex objects and edges are the links between the objects. Due to the large size of such databases, the queries are specified using wildcards allowing retrieval of subgraphs even with a partial knowledge of whole graph (Fig. 1.3 (b)).

Besides storing data in graphs, many of the aforementioned applications require tools to compare different graphs, recognize different parts of graphs, and retrieve and classify

them. In the context of non-structured data (such as strings), modern search engines answer keyword-based queries extremely efficiently. This impressive speed is due to clever inverted index structures, caching, and the utilization of parallel and distributed computing. Just as keyword searching matches words against sequences, keygraph searching matches query graphs against underlying data graphs. Many efforts have been done to generalize keyword search to keygraph searching, but such generalization is not so natural: Keyword searching has *polynomial* complexity on the database size. Keygraph searching has *exponential complexity*, and thus belongs in a entirely different class of problems. More precisely, by keygraph searching we refer to one of the following problems:

1. *Isomorphisms or Exact Match*: Given a query graph G_a and a data graph G_b , we want to determine if two graphs are the same: whether G_a and G_b are isomorphic and whether we can map the vertices of G_a to be vertices of G_b, maintaining the corresponding edges in G_a and G_b. This problem is known to be in NP but it is still not known whether is in P or in NP-complete [83, 80].

2. *Subgraph Isomorphism or Subgraph Exact Matching*: Given a query graph G_a and a data graph G_b, we say that G_a is subgraph isomorphic to G_b if G_a is isomorphic to a subgraph of G_b. Note that G_a may be subgraph isomorphic to several subgraphs of G_b. This problem is known to be in NP-complete. Moreover, finding all the subgraphs matching the query graph G_a (i.e. all the occurrences of G_a in G_b) is more expensive than finding just a single occurrence.

3. *Subgraph Matching in a Database of Graphs*: Given a query graph G_a and a database of data graphs D, we want to find all the occurrences of G_a in each graph in D. Although graph to graph matching algorithms can be used (the solutions of the problem 2), efficiency reasons dictate the use of special techniques in order to reduce the search space in the database and the time complexity. Obviously, this problem is also in NP-complete.

A simple enumeration algorithm to find the occurrences of a query graph G_a in a data graph G_b, is to generate all possible maps between the vertices of the two graphs and to

check whether each generated map is a match. The complexity of such an algorithm is exponential, but it is the best known algorithm.

There have been many attempts to reduce the combinatorial cost of graph searching. Research efforts have taken three directions: the first is to study matching algorithms for particular graph structures (planar graphs [96], bounded valence graphs [123] and associated graphs [19, 119]); the second is to figure out tricks to reduce the number of generated maps [181, 140, 50]; and the third is to provide approximate algorithms which have polynomial complexity but they are not guaranteed to find a correct solution [6, 44, 73, 84, 182, 194].

Several algorithms have been developed for keygraph searching (problems 1 and 2) for cases in which exact matches are hard to find [29, 31, 71, 75, 121, 51, 135, 179, 93]. Such algorithms are useful in applications dealing with noisy graphs. These algorithms employ a cost function to measure the similarity of the graphs or equivalently to transform one graph into another. For example, a cost function may be defined based on *semantic transformations* which depend on the specific application domain and allow the matching of vertices with discordant values; and on *syntactic transformations* (branch insertions and deletions) which match structurally different parts of the graphs. (Of course, approximate algorithms can also be used for noisy data graphs.)

In the context of query graphs in a database of graphs (problem 3) most of the existing methods are designed for specific applications. For a review see [188, 174] and chapter 3. Several querying methods for semi-structured databases have been proposed [128, 46, 166, 174, 157]. Moreover, commercial products [97, 18] and academic projects [103] exist for subgraph searching in biochemical databases. These two examples of applications have a different underlying data model (in the first the database is seen as a large graph while in the second the database is a collection of graphs), but they have the following common techniques: regular path expressions([193, 46]) and indexing methods are used during query time to locate substructures in the database and to avoid unnecessary traversals of the database, respectively.

In contrast to application-specific methods, there are only a few application-independent methods for querying graph databases with the data graph and the query graph having the ”same size” ([36, 158],[155, 169]); methods in which the “same size” restriction is relaxed can be found in [131, 160].

A common idea in the above algorithms is to index the similarities between the graphs or the subgraphs of the database and organize them in suitable data structures. Messmer and Bunke [130, 131] proposed a method that indexes the labelled graphs of a database in exponential time and computes a subgraph isomorphism in polynomial time. Both indexing and matching are based on all possible permutations of the adjacent matrices of the graphs. The performance of the above algorithm has been improved in [132] where only a set of possible permutation is maintained. Cook et al. [49] proposed a different approach (not based on indexing techniques) for subgraph searching of a database. They found similar repetitive subgraphs in a single-graph database applying standard graph matching algorithms.

1.1.1 GraphGrep

In this thesis we present an application-independent method to perform *exact* subgraph queries in a database of graphs. Our system, GraphGrep [160], finds *all* the occurrences of a graph in a database of graphs. Close to the underlying ideas in [97, 132], the search algorithm is based on an indexing technique which classifies small substructures of the database graphs.

GraphGrep assumes that graph vertices have an identification number (id-vertex) and a label (label-vertex). Here we assume that graphs are labeled only in the vertices . An *id-path* of length n is a sequence of $n + 1$ id-vertices with a binary relation between any two consecutive vertices. Similarly a *label-path* of length n is a sequence of $n+1$ label-vertices.

The indices are called the fingerprints of the database. They are generated during a preprocessing step of the database and they are viewed as an abstract representation of the structural features of the graphs. The fingerprints are implemented using a hash table: each

row of the hash table contains the number of id-paths that are associated with the label-path hashed in this row. The label-paths have length zero and up to a constant value lp. A suitable value for lp allow us to perform the pre-processing of the graphs in polynomial time. The id-paths created during fingerprinting are not discarded but they are stored in tables; each table corresponds to a different label- path. In order to find a matching for the query, an algebra handles the information in those tables.

To formulate queries we introduce a graph query language which we term Glide: Graph LInear DEscription language. Glide descends from two query languages Xpath [46] for XML documents and Smart-Smiles [97] for molecules. In Xpath, queries are expressed using complex path expressions where the filter and the matching conditions are included in the notation of the vertices. Glide uses graph expressions instead of path expressions. Smiles is a language designed to code molecules and Smart is a query language to discover components in a Smiles databases. Glide borrows the cycle notation from Smiles and generalizes it to any graph application.

The performance of the algorithm has been tested on NCI databases [137] of size up to $120,000$ molecules and on random databases. We have compared GraphGrep with the best known software, (Daylight and Frowns) and we have obtain very promising results. A software implementation of GraphGrep and its demo are publicly available at (www.cs.nyu.edu/shasha/papers/graphgrep).

1.2 Searching in Trees

Many applications use trees, which are particular types of graphs, to describe data. There are several tools for searching, indexing, storing and retrieving subtrees in a collection of trees. By *keytree searching* we refer to special graph and subgraph matching algorithms for rooted trees.

- We first consider an database of ancient coins. The traditional way of sharing information about ancient coins or other precious old items between archaeological institutes and Museums is through collections of photographs. Typically whenever

a new coin is found an expert uses his previous knowledge and all the information known about the object, to find its origin and its classification. In this difficult task, accessing to catalogues of coins, such as the one described above, is an invaluable tool to validate or reject archeological hypotheses. However, the size and number of catalogues available for consultation are limited. Computers are changing this traditional framework in several ways. First, fast and cheap scanning techniques are greatly increasing the volume of available pictures. Second, "intelligent" methods provide valuable support. As the size of image databases grows the traditional methods of finding an image become nearly useless. In this context, algorithms to analyze and compare efficiently thousands of pictures signal a significant breakthrough ([138, 92, 108, 101, 76]). A coin semantically is as a complex object with a precise syntax and structure. An expert defines the syntax and the structure of a coin locating the most important features that are helpful in assessing its identity. These features can be organized in a tree ([139]) and the distance between two trees (expressed as the cost to change one tree in another [142]) provides a heuristic way to estimate the distance between the corresponding coins. A partial features tree associated with a generic coin is shown in Fig. 1.4. The organization of such a tree largely depends on a thoughtful analysis that an information technologist carries out together with an expert Archeologist. A general rule is that the nodes of a tree in the higher levels are associated with the more selective features. This rule may be overridden whenever it would create inefficiency in dealing with the resulting tree structure.

- XML [197] is the emerging standard language to share and represent information on the Web. The natural structure of a XML data is a graph due to the particular reference mechanism between its elements. Ignoring such references, an XML document becomes an ordered tree (Fig. 1.5). Most of the database systems for XML [198, 20, 17, 120] have chosen trees as underlying data model.

- Natural language processing applications (such as retrieval of information in digital libraries [72, 136] and searching for matches in example-based translation systems

[141, 149]) use trees to represent statements or document layouts. In [142], hierarchical structures of trees describe the syntactic rules to construct English sentences (Fig. 1.6). In [72], in order to answer queries such as ”find all the pages containing the title adjacent to an image”, trees describe the geometrical features of document pages. Here, the hierarchical structures of trees represent a subdivision in regions of the pages where a region in a page is a text or an image between columns and white spaces.

The preceding applications share some characteristics. The database can be represented as a single tree ([118, 171, 173]) or as a set of trees ([20]). Moreover, each tree could be *ordered*, in which case the order among siblings is significant (as in the XML data model) or could be *unordered* as in hereditary trees and some archeological databases. Searching may require, like graphs, finding exact or “approximate” tree and subtree matchings. One way to be precise on *approximate* is by counting the number of paths in the query tree that do not appear in the data tree [162]; other distance functions are possible [40, 41, 42, 170, 190, 198, 202]. By approximate tree matching we also mean the matching with query trees containing wildcards [133].

Depending on the tree structure, keytree searching problems have complexity ranging from linear (P) to exponential (NP-complete) on the tree size. More precisely, solving an exact tree or subtree matching problems is polynomial in both unordered and ordered trees [94, 112, 63],[37, 39, 47, 48],[178, 127, 156],[61, 185, 90],[89, 88, 87],[64, 124],[186, 147]. Approximate tree searching belongs in NP-complete class for unordered trees and in P for ordered trees [176, 122, 154, 105, 183, 201, 163, 202].

In order to improve query processing time, considerable research efforts [14, 25, 35, 26, 45, 161, 154, 180, 200, 142], have prompted to combining several sophisticated data structures with approximate tree matching algorithms that work over generic metric spaces (i.e. algorithms that do not depend on special properties of the distance function considered).

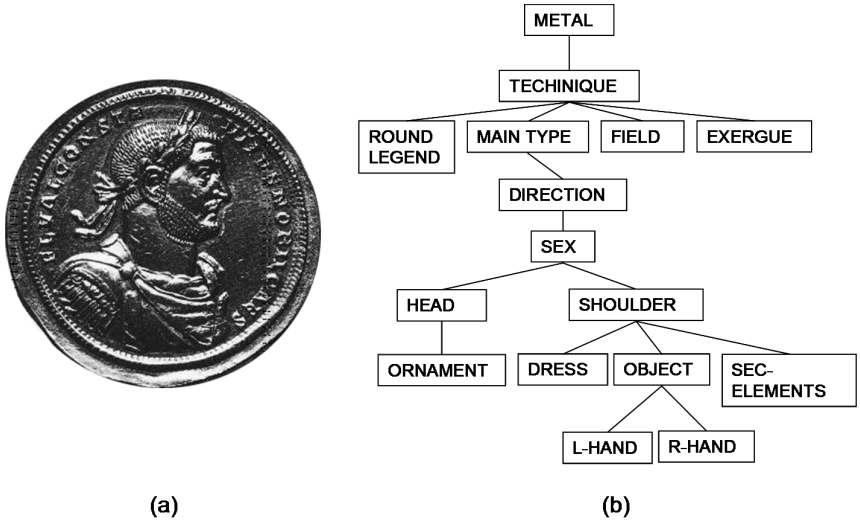


Figure 1.4: (a) A late Roman Empire coin. (b) General tree representation of a coin. METAL: {Gold, Silver, Bronze,Brass, Copper}, TECHNIQUE {Cast, Struck}, DIRECTION:{Left, Right, Facing}, SEX:{ Male, Female}, HEAD:{ Bare, Radiate, Crowned, Laureate, Veiled, Helmeted}, ORNAMENT:{Diadem, Bare, Laureate, Feather}, DRESS:{ Draped, with a Cuirass, Draped and with a Cuirass}, SEC.ELEMENTS:{Shield, Crescent}

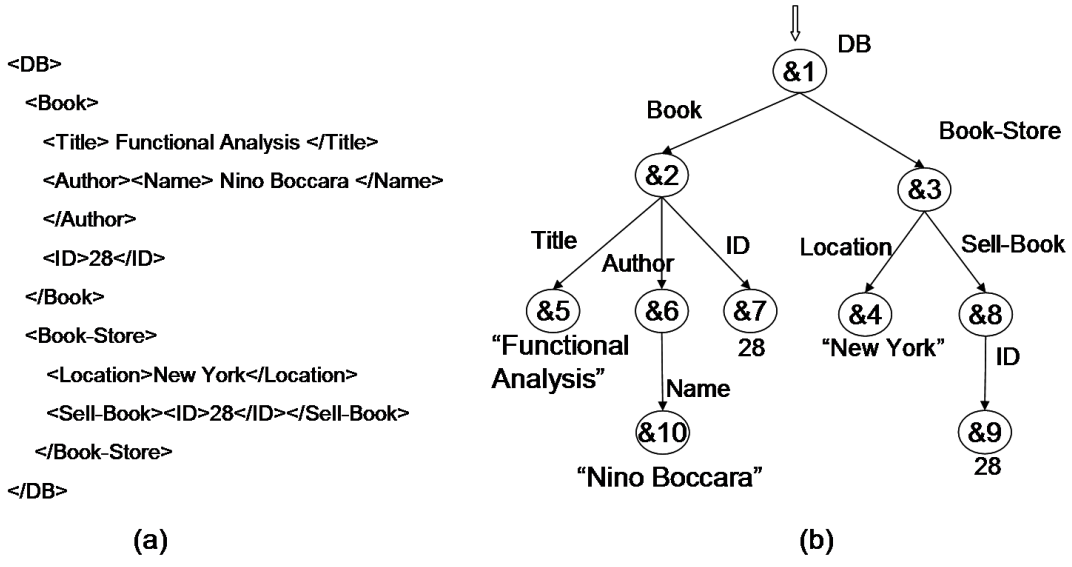


Figure 1.5: (a) A XML document. (b) A XML tree.

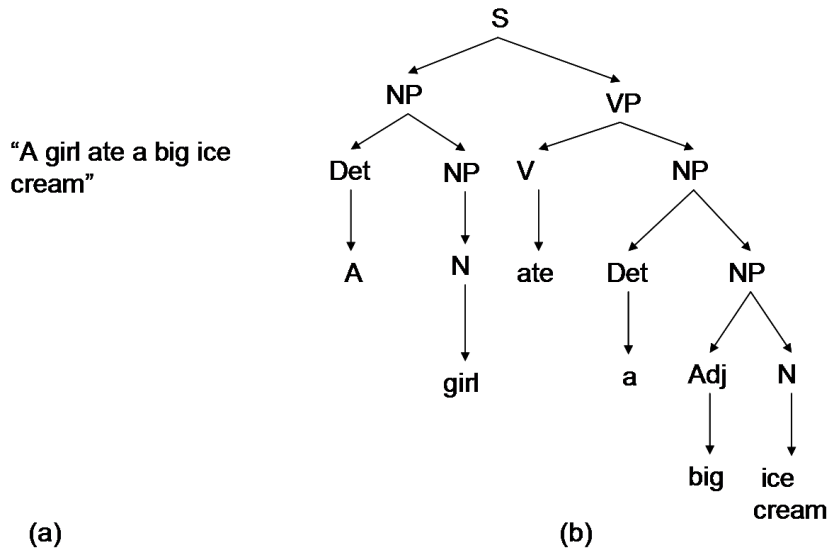


Figure 1.6: (a) An English sentence. (b) A tree describes the syntactic rules of the sentence.

Certain tree searching methods [35] make use of properties of the underlying distance function between the objects in the database. Others only assume that the distance function is a metric. This is the case in the Fixed Query tree (FQ-tree) algorithm [14], the Vantage Point tree (VP-tree) algorithm [43, 200], its upgraded variation (known as) Multiple Vantage Point tree (MVP-tree) algorithm [25], and the M-tree algorithm [45].

The method presented in this thesis inscribes itself naturally in the family of distance-based index structures described above. It is based on an approximate searching technique for a collection of trees introduced by Oflazer [142]: To retrieve trees whose distance from a given query is below a given threshold, each tree is, in turn, coded as a sequence of paths from its root to every leaf. The sequences of paths describing all the trees are eventually compactly stored in a trie. The trie data structure naturally provides a way to interrogate the database with error tolerant and approximate queries: an obvious strategy is to traverse the trie from the root to the leaves. In this search it is possible to do an early pruning of a branch as soon as it is seen that it leads to trees whose distance from the query exceeds the given threshold.

We improve the above method by proving and exploiting a triangle inequality property for Oflazer's distance. In particular, we propose a saturation algorithm for the trie. The proposed procedure grants earlier pruning of unsuccessful branches and hence greatly improves search efficiency. The speed-up at query time is obtained by learning some a priori information about the database structure. In particular a key step for the new procedure is the pre-computation of the mutual distances between elements in a subset of the database. For the sake of simplicity, in this thesis we pre-compute all possible distances between pairs of objects in the collection and use the resulting symmetrical matrix of distances. The technique works equally well by precomputing all the distances only relative to a randomly chosen object [35], or relative to several chosen objects [154], or relative to an arbitrary set of pairs of objects in connection with a Floyd-Warshall style algorithm [161]. Since the triangle inequality property is especially useful between far away objects, the mutual distance may be calculated only between those objects that are far a part in the trie. The triangular property would be applied in this case only on these elements to further pruning the search space. These alternatives to the computation of the complete distance matrix are important especially if the technique has to be applied to very large databases: in this case the whole distance matrix is too large to be stored in primary memory at once.

The precomputed distance information is used as follows: suppose that while traversing the trie a node N with satisfies $d(Q, N) > t$ is reached. Here Q denotes the query and t a positive, user selected threshold. The triangle inequality allows us to estimate the distance between Q and another node M in the trie. More precisely if $d(Q, N) - d(N, M) \geq t$ then $d(Q, M) \geq t$ and node M can be skipped. The pruning may be enhanced by applying a recursive procedure which will eliminate all trie nodes whose descendants have been already discarded.

In [76, 139] we have shown that the original retrieval technique introduced by Oflazer [142] for large collection of statements in natural language, can be successfully applied to databases of structured images such as stamps and coins. Here, we report the results obtained with a collection of up to 300 postal stamps and compare them with the results obtained with other retrieval techniques (particularly with Oflazer's original technique and

with the MVP-trees approach).

1.3 Searching in Temporal Probabilistic Object Data Model

In the second part of the thesis we explore another aspect the next generation databases: the formulation of a *temporal, probabilistic object data base*.

Object data models [2, 58, 150, 12] have been used to model numerous applications ranging from multimedia applications[54, 55], financial risk applications[38], and logistics and supply chain management applications [5], weather applications [60] as well as many others. Many of these applications naturally need to represent and manipulate both time and uncertainty.

- We first consider a transportation logistics application [24, 5]. A commercial package delivery company (such as UPS, Fedex, DHL, etc.) has detailed statistical information on how long packages take to get from one zip code to another, and often even more specific information (e.g., how long it takes for a package from one street address to another). A company expecting deliveries would like to have some information about when the deliveries will arrive of the form "the package will be delivered between 9 am and 1pm with a probability between 0.1 and 0.2, and between 1pm and 5pm with a probability between 0.8 and 0.9" (here, probabilities are degrees of belief about a future event, which may be derived from statistical information about similar previous events). Such an answer is far more helpful to the company's decision making processes than the bland answer "It will be delivered sometime today between 9 am and 5pm". For example, it helps them schedule personnel, prepare receiving facilities (for hazardous material and other heavy materials), prepare future production plans, etc. In addition, the different entities involved in such an application are typically stored using object models — this is because different vehicles (airplanes, trucks, etc.) have different capabilities, and because different packages (letter, tube, hazardous material shipments for commercial customers, etc.) have widely varying attributes. The shipping company itself has extensive need for such data. For example, the company would need to query this database to create plans

that optimally allocate and/or use available resources (space on trucks, personnel, etc.) based on their expectations of the probable workload at future time points.

- Weather database systems (such as the Total Atmospheric and Ocean System or TAOS system developed by the US Department of Defense) use object models to represent weather data. Time and uncertainty are omnipresent in weather models and most decision making programs rely on knowledge of this kind of uncertainty in order to make decisions.
- There are numerous financial models [8, 38], which banks and institution lenders use to attempt to predict when customers will default on credit. Such models are complex mathematical models involving probabilities and time (the predictions specify the probability with which a given customer will default over a given period of time). Furthermore, models to predict bankruptcies and loan defaults vary substantially depending upon the market, the type of credit instrument (consumer credit card, mortgage, commercial real estate loan, HUD loan, construction loan, etc.), the variables that affect the loan, various aspects about the customer, etc. Such models are naturally represented via object models, and time and uncertainty parameterize various features of the model.

There is extensive work in the literature on temporal databases and temporal object-oriented databases ([143, 98, 177, 168]). Probabilistic extensions of relational databases are also well-explored in the literature; see especially [117, 69] for more background and a detailed discussion of recent work on probabilistic relational databases. Recently, more complex data models have been extended by probabilistic uncertainty in a number of papers. In particular, Eiter et al. [69] presented an approach that adds probabilistic uncertainty to complex value relational databases, while Kornatzky and Shimony [110, 111] and Eiter et al. [68] described approaches to probabilistic object-oriented databases. There is further work on non probabilistic temporal indeterminacy in databases. In particular, Snodgrass [167] models indeterminacy using a model that is based on a three-valued logic. Moreover, Dutta [66] and Dubois and Prade [65] propose a fuzzy logic approach to temporal indeterminacy, while Koubarakis [114, 113] and Brusoni et al. [27] suggest approaches based on

constraints. Finally, Gadia et al. [82] introduce partial temporal databases, which are based on partial temporal elements.

There is, however, very little work on the integration of temporal reasoning and probabilistic databases. In particular, Dyreson and Snodgrass in their pioneering work [67] — an extension of the *SQL data model and query language* by probabilistic uncertainty on time points — and subsequently Dekhtyar et al. [57] presented approaches to temporal indeterminacy in relational databases based on probabilistic uncertainty.

In this thesis, we propose a theoretical foundation for object databases that allow representations of temporal data and probabilistic uncertainty and thus probabilistic temporal indeterminacy. Our approach is a temporal extension of the model by Eiter et al. [68]. We also introduce a new implicit data model and an implicit algebra which correctly implements its explicit counterpart, and can be efficiently realized. Moreover, several algebraic operations are brand new. As in the case of [117, 68, 69], our model uses interval probabilities rather than point probabilities. The reasons are three-fold: First, probability intervals are a generalization of point probabilities, and thus can handle all point probabilities. Second, they allow expression of imprecise probabilistic knowledge. For example, it is possible to say that an event occurred at time t with probability p with a margin of error δ, denoting that the probability of e occurring at time t lies in the interval $[p-\delta, p+\delta]$. Hence, probability intervals are better suited especially to represent statistical knowledge and subjective degrees of belief ([116]). Third, even if we know a point probability for two events e_1 and e_2, in general we can only infer a probability interval [23] for the conjunction and disjunction of e_1 and e_2 (unless we make additional assumptions such as independence). We refer the reader to Chapter 8 for a study in depth of the related work on temporal and probabilistic data model and the comparison with our approach.

The main contributions of the present work can be summarized as follows:

- We define the concept of a temporal probabilistic object base (TPOB for short). In particular, we introduce the notion of a TPOB-schema, and we define the important concept of (explicit) TPOB-instances over a TPOB-schema. Such TPOB-instances represent a probabilistic statement over a set of time points by simply enumerating all time points along with their probability intervals.

- We define algebraic operations that operate on TPOB-instances. We define selection, restricted selection, renaming, projection, extraction (a new operation), natural join, Cartesian product, conditional join, and the set operations of intersection, union, and difference. We remark that the operations of projection and Cartesian product are simply extensions of their counterparts from the classical relational algebra, while all the other operations are full-fledged complex-object operations (as they address the "inner" components of object values).
- We introduce the notion of consistency for TPOB-instances and a related notion called coherence and show that under appropriate assumptions, all our algebraic operations preserve consistency and coherence of schemas and instances, respectively.

- The TPOB-instances described above are *explicit* in the following sense. Given an event e, for each time point t, an explicit TPOB-instance $\mathbf{I}$ specifies a probability interval describing the probability with which event e occurred (or will occur) at time t. Though explicit TPOB-instances make it easy to formally define the various algebraic operations, they can sometimes be very inefficient from a space point of view (and hence also inefficient from the point of view of computing algebraic operators). For example, if a company wants to record that a particular shipment will arrive sometime between 8am and 5pm on day D, and the temporal granularity used is seconds, then we need to make this statement for each of the $9 \times 60 \times 60 = 32,400$ seconds between 8am and 5pm. In order to avoid this problem, we define the concept of an implicit TPOB-instance. Implicit TPOB-instances represent a probabilistic statement over a set of time points by a probability distribution function in combination with a constraint, which specifies the set of time points. Hence, implicit TPOB-instances allow for an efficient implementation of algebraic operations, while explicit TPOB-instances make defining algebraic operations relatively transparent.
- We show that there is a translation ε that maps an implicit TPOB-instance $\mathbf{I}$ to an explicit one $\varepsilon(\mathbf{I})$. For each algebraic operation on explicit TPOB-instances, we define a counterpart on implicit TPOB-instances. As they work on succinct representations,

Figure 1.7: Correctness theorems: commutativity diagram

they have better computational properties.

- We show that the algebraic operations on implicit TPOB-instances correctly implement their counterparts on explicit TPOB-instances (that is, the answers produced by the operations on implicit TPOB-instances succinctly represent the answers produced by the operations on explicit TPOB-instances). Figure 1.7 provides a diagrammatic representation of what these results look like for unary operators (a similar figure can be shown for binary operators). For unary algebraic operators op, the correctness theorems are of the form $\varepsilon(op^i(\mathbf{I})) = op^e(\varepsilon(\mathbf{I}))$. We use op^i (resp. op^e) to denote the implicit (resp. explicit) versions of the operator op. A similar kind of correctness result holds for binary algebraic operators.

1.4 Organization

This thesis is organized in two parts. In the first part we focus on searching algorithms for graph and tree databases. In the second part we present data structures for temporal and probabilistic object databases. Bellow we give a detailed description of the chapters:

Part I. In Chapter 2, we briefly review the notation and some basic facts on graph and tree data structures. (Readers familiar with the subject may skip this chapter.)

In Chapter 3 we report the state-of-art for algorithms for graph and tree matching and for searching in tree and graph databases. In Chapter 4, we present Glide, a graph query language, and we compare it with similar languages. In chapter 5 we present GraphGrep, a method for querying graph databases. We describe indexing and matching techniques. We give performance results on very large databases of molecules and on random graph databases. We also report results from comparisons of GraphGrep with Frowns[102], a software package for searching in molecule databases. In Chapter 6. We also present an optimization method for indexing techniques which are used for fast retrieval of subtrees

from a database of trees, and we compare it with existing methods. With Chapter 7, we conclude the first part of the thesis with a discussion on directions for future work.

Part II. In the second part of the thesis we present the model and the algebra for a probabilistic, temporal object data base. In Chapter 8 we overview related work. In Chapter 9 we introduce some basic definitions. In Chapter 10 we present the notions of TPOB-schemas and of explicit TPOB-instances. In Chapter 11 and 12, we introduce the unary and binary algebraic operations on explicit TPOB-instances. In Chapter 13, we show that these algebraic operations preserve coherence and consistency of schemas and instances. In Chapters 14 and 15 we define implicit TPOB-instances and the algebraic operations on implicit TPOB-instances, respectively. Finally, in Chapter 16 we conclude with a discussion on future work.

Part I

Graph and Tree Databases Searching Based on Indexing Techniques

Chapter 2

Basic Definitions

In this Chapter we recapitulate some basic definitions.

Definition 2.0.1 (Graph)**.** A graph G is a pair of finite sets (V, E). $V = \{v_0, v_1, \ldots, v_k\}$ is the set of vertices in G. $E = \{(v_i, v_j) \mid i, j \in \{1, \ldots, k\}\}$ is the set of edges in G, i.e. it is the set of connections among the vertices in G.

E may also be defined as a binary relation on V.

Definition 2.0.2 (Undirected Graph)**.** A graph $G = (V, E)$ is called undirected if E consists of unordered pairs of vertices —(v_i, v_j) is the same as (v_j, v_i). The edge (v_i, v_j) is *incident* to both vertices v_i and v_j.

In an undirected graph the *valence* or *degree* of a vertex v is the number of incident edges in v.

Definition 2.0.3 (Directed Graph)**.** A graph $G = (V, E)$ is called directed if the edges have a direction. (v_1, v_2) is not the same as (v_2, v_1). The (v_1, v_2) is an *out-going* edge for v_1 and an *in-coming* edge for v_2.

In a directed graph the *valence* or *degree* of a vertex v is the number of out-going edges from v.

Definition 2.0.4 (Labelled Graph)**.** Let L a set of labels, $\mu : V \to L$ be a node labelling function and $\nu : E \to L$ be an edge labelling function, then $G = (V, E, \mu, \nu)$ is a labelled graph.

$G = (V, E, \mu)$ is a *node labelled* graph and $G = (V, E, \nu)$ is an *edge labelled* graph. If L is a set of integers, real numbers or any type of quantity an edge labelled graph is called weighted graph.

Graphs with structural properties are given special names.

Definition 2.0.5 (Dense Graph)**.** A graph is called dense if the number of edges $|E|$ is $O(|V|^2)$, otherwise it is called *sparse*.

A *complete* graph is an undirected graph in which exists an edge between every pair of vertices—$|E| = |V|^2$.

Definition 2.0.6 (Planar Graph)**.** A planar graph is a graph that can be drawn so that the edges only touch each other where they meet at vertices.

Definition 2.0.7 (Bipartite Graph)**.** A bipartite graph is a graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$.

Definition 2.0.8 (Subgraph)**.** A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \in V$ and $E' \in E$.

A complete subgraph of G is a *clique*.

Definition 2.0.9 (Path in a Graph)**.** A path of length k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $(v_0, v_1, \ldots, v_k)$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$.

The *length* of the path is the number of edges in the path.

If there is a path p from u to u' we say that u' is reachable from u via p.

A path is *simple* if all vertices in the path are distinct.

In a path $(v_0, v_1, \ldots, v_k)$, for any pair of consecutive nodes as for example $v_0$and v_1, v_0 is a *parent* of v_1 and v_1 is a *child* of v_0.

Definition 2.0.10 (Id-path in a Graph)**.** Let $G = (V, E)$ be a graph. Let $id : V \to N$ be a one way function that associates to each vertex an unique number. An id-path corresponding of a path $(v_0, v_1, \ldots, v_k)$ in G is a sequence $(id(v_0), id(v_1), \ldots, id(v_k))$.

Definition 2.0.11 (Label-path in a Graph)**.** Let $G = (V, E, \mu)$ be a node-labelled graph. A label-path corresponding to a path $(v_0, v_1, \ldots, v_k)$ in G is a sequence $(\mu(v_0), \mu(v_1), \ldots, \mu(v_k))$.

Definition 2.0.12 (Acyclic Graph)**.** A graph is *acyclic* if all its possible paths are simple.

Definition 2.0.13 (Graph Isomorphism)**.** Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. In other words, we can relabel the vertices of G to be vertices of G', maintaining the corresponding edges in G and G'.

Definition 2.0.14 (Subgraph Homomorphism)**.** Given two graphs $G = (V, E)$ and $G' = (V', E')$ then G is subgraph homomorphic to G' if G is isomorphic to a subgraph of G'.

Definition 2.0.15 (Subgraph Isomorphism or Subgraph Matching)**.** A graph $G = (V, E)$ is subgraph isomorphic to $G' = (V', E')$ if there exists a $f : V \to V'$ such that $(u, v) \in E$ only if $(f(u), f(v)) \in E'$. In other words, we can relabel the vertices of G to be vertices of G', maintaining the corresponding edges in G. Note that G may be subgraph isomorphic or homomorphic to several subgraphs of G'.

Next we report several definitions about kinds of graphs called trees. In the trees the vertices are usually called nodes.

Definition 2.0.16 (Tree)**.** An acyclic and undirected graph is called a *free-tree*. A free-tree with a distinguished node, named root, is a rotted tree. The root node does not have a parent node.

We call a tree a rotted tree. In a tree, nodes without children are called *leaf*.

A non leaf node is an *internal* node.

If two nodes have the same parent, they are siblings.

Definition 2.0.17 (Height of a Tree)**.** The length of a path from the root of the tree T to a node v is the *depth* of v in T. The largest depth of any node in T is the height of T.

Consider a node v in a tree T with root r. Any node u on a path from r to v is called an *ancestor* of v. If u is an ancestor of v, then v is a *descendant* of u.

Definition 2.0.18 (Subtree)**.** A subtree of a tree T rooted at node v is the tree induced by descendants of v in T.

Definition 2.0.19 (Ordered Tree)**.** A tree is ordered if the order of the children of each node is important– first child, second child and so on. Given two nodes with the same children but in a different ordered the subtree induced by those are different.

Definition 2.0.20 (Positional Tree)**.** A positional tree is a tree in which the children of a node are labelled with distinct positive integers. The ith child of a node is absent if no child is labeled with integer i.

Chapter 3

State of Art

Keygraph (or keytree) searching in databases consists of three basic steps:

- *Reduce the search space by filtering*. For a database of graphs a filter finds the most relevant graphs; for a single-graph database it identifies the most relevant subgraphs. In this work, we restrict our attention to filtering techniques based on the structure of the labeled graphs (paths, subgraphs). Since looking for subgraph structures is quite difficult, most algorithms choose to locate paths of vertex labels.

- *Formulate query into simple structures*. The keygraph can be given directly as a set of vertices and edges or as the intersection of a set of paths. Furthermore the query can contain wildcards (representing vertices or paths) to allow for more general searches. This step normally reduces the query graph to a collection of small paths.

- *Match*. Matching is implemented either by traditional (sub)graph-to-graph matching techniques, or by combining the set of paths that result from processing the path expressions in the query through the database.

The algorithms for searching in graph databases and tree databases are described in two distinct Sections; 3.1 and 3.3, respectively. The complexity of the (sub)graph-to-graph matching problem and a review of certain algorithms with potential applications to keygraph searching in databases are discussed in Section 3.2. Finally, in Section 3.4 we

describe some techniques for the (sub)tree-to-tree matching and for the tree embedding problem.

3.1 Keygraph Searching in Graph Databases

Cook *et al.* [49, 62] (SUBDUE) applied an improvement of the inexact graph matching algorithm proposed in [32] (which is based on A^* algorithm [140]) to find similar repetitive subgraphs in a single-graph database. These methods are of interest primarily for the matching step. SUBDUE has been applied to discovery and search for subgraphs in protein databases, image databases, Chinese character databases, CAD circuit data and software source code. Furthermore an extension of SUBDUE, WebSUBDUE [126], has been applied to hypertext data.

Guting [91] proposed a general purpose object-oriented data model and query language (GraphDB) for graph databases. Vertices in a graph are classes representing data (objects) and edges are classes linking two vertices. GraphDB contains classes to store several paths in the database. Path classes and indexing data structures (e.g. B-tree, LSD) are used to index vertices, paths and subgraphs in the graph database. Graph queries are specified using regular expressions and they may restrict the search space to a subgraph of the whole graph. GraphDB provides graph search operations to find shortest paths between two vertices or to find subgraphs from a starting vertex within a distance range. The implementation is based on A^*.

Daylight [97] is a system used to retrieve substructures in databases of molecules where each molecule is represented by a graph. Daylight uses fingerprinting to find relevant graphs from a database (step 1). Each graph (of the database) is associated with a fixed-size bit vector— called the fingerprint of the graph. Given a graph G, its fingerprint bits are set in the following way: all the paths in G of length zero and up to a limit length are computed; each path is used as a seed to compute a random number; and the bit representation of this number is added to the fingerprint. The fingerprint represents structural features of the graph. The similarity of two graphs is computed by comparing their fingerprints.

Some similarity measures are: the Tanamoto Coefficient (the number of bits in common divided by the total number); the Euclidean distance (geometric distance); and the Tversky similarity—used to measure the similarity of a query graph with a subgraph of a data graph. The search space is filtered by comparing the fingerprint of the query graph with the fingerprint of each graph in the database. Queries can include wildcards. For most queries, the matching is implemented using application specific techniques. However queries including wildcards may require exhaustive graph traversals.

Widom et al. [3, 85, 128] proposed a system, called Lore, to store and query a semistructured database (which is modeled as a large rooted labeled directed graph; see [1, 174, 187] for a survey). Lore uses four kinds of indices to accelerate (regular) path expression searching. For each edge label l in the graph, a value index (Vindex) is used to index all the vertices that have incoming edges labeled with l and with atomic values that satisfy some condition. A text index (Tindex) is used for all vertices with incoming l-labeled edges and with string atomic values containing specific words. A link index (Lindex) indexes the vertices with outcoming l-labeled edges. A path index (Pindex–DataGuide) indexes all the vertices reachable from the root through a labeled path. Each path query that starts at the root uses the DataGuide. All other path queries use the other three indexes in which case they find a set of candidates and then traverse the graph to prune away paths that do not match the query path. Because the other indexes are unselective, there are potentially many more candidates than matching paths.

Milo and Suciu [134] proposed a data structure, called T-index, to index semistructured database vertices that are reachable from several regular path expressions. A T-index is a non-deterministic automaton whose states represent (roughly) the equivalence classes produced by the Rabin-Scott algorithm and whose transitions correspond to edges between objects in those classes. By relaxing the determinism requirement imposed on DataGuides, a T-index can be constructed and stored more efficiently. They may represent a more efficient DataGuide in both time and space. For example the authors reported that in a graph of 1500 vertices , the T-index size is 13% of the size of the graph database.

Bunke and Messmer [131, 130] proposed using a tree to organize a graph database:

Every possible permutation matrix of a graph in the database is represented by a path in the tree—from the root down to a leaf. Each edge in a path represents a vertex in a graph and stores information about the connectivity of the vertex with its ancestor vertices along the path. A keygraph search is computed by a traversal of the decision tree. Despite the fact that the keygraph search is computed in polynomial time on the size of the query graph, the computation of the decision tree requires exponential time on the size of the graphs in the database. It can be applied to a database of 30 graphs where each graph has up to 11 vertices.

Similar to the tree representation, Bunke and Messmer [132] proposed preprocessing the graphs in the database to deduct a hierarchy of a small set of their possible subgraphs (common subgraphs are represented only once). During query time, a bottom up traversal of the hierarchy matches the query graph with subgraphs of the data graphs (matching uses a divide-and-conquer approach—from the single vertices to the data graphs). This process requires a consistent number of subgraph matchings between components of the query graphs and the subgraphs of data. Usually, its efficiency depends on the order which the graphs in the database are taken during the preprocessed time.

The Laplacian matrix of a graph is an adjacent matrix of the graph that also stores the valence of each vertex. Sossa and Horaud [169] computes for each graph the second immanantal polynomial of its Laplacian matrix. Experimental evidence suggests that the most discriminant coefficient is the second last. This coefficient along with the first one (which encodes the size of the graph) are used to index the graph and to retrieve graphs. This method can only compare graphs of the same size. There is a one-to-one correspondence between a 12-vertex graph and the coefficients of its immanantal polynomial. Based on this observation, the authors propose a hierarchical encoding by the decomposition of a graph into subgraphs. In this way the database filtering is based on subgraphs of the query graph.

GraphGrep, presented in Chapters 1 and 5 (see also [160, 164]), is a new hash-based algorithm for finding *all* the occurrences of a query graph in a database of graphs. A set of intersecting regular path expressions is deduced by the query graph. GraphGrep uses variable length paths (that may contain cycles) to index the database: this allows efficient

Table 3.1: Comparison of tree and graph searching systems. Filter-out is divided into Item (whether an entire tree or graph can be removed from consideration when matching a given query) and Sub-Item (whether relevant portions of selected trees and graphs are identified by the filtering steps). We use AD for an "Application Dependent" matching algorithm (e.g. tailored for molecules), PC for "Path Combination" (e.g. intersecting paths), and PC-T for a "Path Combination" matching algorithm which requires tree or graph "Traversal". Different systems have different expressive power using wildcards.

System	Application	Database Model	Path-Indexing		Filter-out		Wild-cards	Match
			From	Length	Item	Sub Item		
ATreeGrep	General	Many Trees	Every Leaf	Full	Yes	No	Yes	PC
XISS	XML	Many Trees	Every Node	1	Yes	Yes	Yes	PC
Daylight	Molecule	Many Graphs	Every Node	Parameter	Yes	No	Yes	AD
Frowns	Molecule	Many Graphs	Every Node	Parameter	Yes	No	Yes	VFlib
GraphDB	General	One Graph	Some Nodes	Parameter	Yes	Yes	Yes	PC-T A^*
GraphGrep	General	Many Graphs	Every Node	Parameter	Yes	Yes	Yes	PC
Lore	Semi-structured-Data	One Rooted-Graph	Root Every Node	Full 1	No	Yes	Yes	PC-T
SUBDUE	General	One Graph	No	No	No	No	No	A^*
T-index	Semi-structured-Data	One Rooted-Graph	Root Some Nodes	Full Variable	-	Yes	-	-

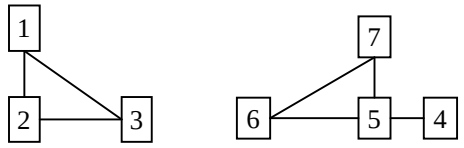
filtering by directly selecting the most relevant subgraphs of the most relevant graphs.

Table 3.1 summarizes the features of several keytree (described in Section 3.3) and keygraph searching techniques for tree and graph databases, respectively.

3.2 Subgraph Matching

In [199] Yannakakis surveyed traditional graph searching problems with applications to data management, including computing transitive closures, recursive queries, and the complexity of path searching in databases. Moreover, we refer to [99, 30, 164] for surveys on graph matching algorithms and applications. We report here some classical subgraph matching techniques, mostly having to do with step 3 of our query processing framework.

A simple theoretical enumeration algorithm to find the occurrences of a query graph G_a in a data graph G_b (Figure 3.1), is to generate all possible maps between the vertices of the two graphs and to check whether each generated map is a match. All the maps can be represented using a *state-space representation* tree: a node represents a pair of matched vertices; a path from the root down to a leaf represents a map between the two graphs. A path from a node at the k_{th} level in the tree up to the root represents a *partial matching* between the graphs; only a subset (k) of vertices have been matched. Only certain leaves correspond to a subisomorphism between G_a and G_b (Figure 3.2). The complexity of such an algorithm is exponential—the problem of subgraph isomorphism is proven to be NP-complete [83].



(a) (b)

Figure 3.1: (a) A query graph G_a. (b) A data graph G_b.

There have been many attempts to reduce the combinatorial cost of query processing in graphs. They can be classified as *approximate, inexact, and exact algorithms*. We recall that approximate algorithms have polynomial complexity but they are not guaranteed to find a correct solution. Exact and inexact algorithms do find correct answers and therefore have exponential worst-case complexity.

Exact Algorithms. Ullmann [181] presented an algorithm for an exact subgraph matching based on the state space search with backtracking algorithm in [53]. A depth-first search on the state space tree representation depicts the algorithm's progress. When a node (a pair of matching vertices) is added to the tree, the isomorphism conditions are checked in the partial matching. If the isomorphism condition is not satisfied the algorithm *backtracks*

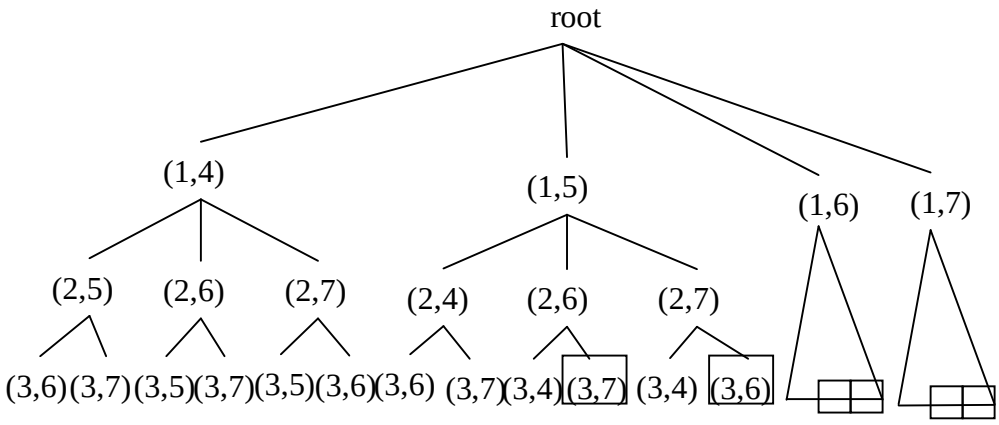


Figure 3.2: All the maps between G_a and G_b. The leaves in the rectangular frames correspond to subisomorphisms between G_a and G_b.

(i.e. the tree-search that would correspond to a full enumeration is pruned.) Upon termination only the paths with length equal to the number of vertices in G_a (corresponding to unpruned leaves) represent a subisomorphism. The performance of the above state-space representation algorithm is improved by a refinement procedure called *forward checking*: in order to insert a node in the tree not only must the sub-isomorphism conditions hold, but, in addition, a possible mapping must exist for *all* the unmatched vertices. As a result, the algorithm prunes the tree-search more efficiently at a higher level (see Figure 3.3(a)).

Based on the Ullmann's algorithm, Cordella et al. [51, 50, 52] proposed a efficient algorithm for graph and subgraph matching using a more selective feasibility rules to cut the state search space. Foggia et al. [79] reported a comparison of the above algorithm with the algorithm of Ullman, McKay [129] and Schimidt et al. [152] showing that, so far, it does not exists one algorithm more efficient of the others on all kind input graphs.

Barrow et al. [19] and Levi [119] show that subgraph matching is equivalent to maximal clique finding in association graphs. An association graph, derived from G_a and G_b, is a graph where each vertex represents a pair of vertices (one from G_a and one from G_b) defined to be in a relation (e.g. with the same label). Given two vertices in the association graph an edge between them exists if the corresponding edges in G_a and G_b exist.

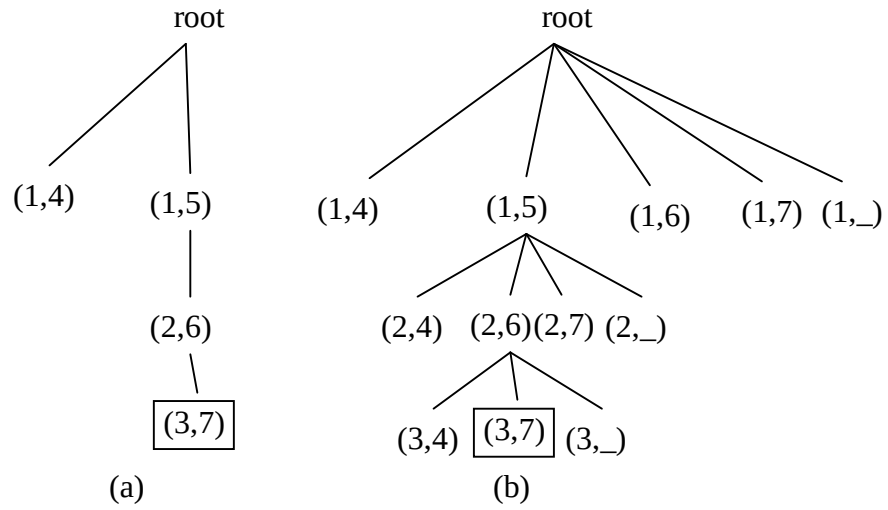


Figure 3.3: The tree search-space to reach the first isomorphism. (a) The tree-search space pruned by Ullmann's algorithm (b) The tree-search space pruned by Nilsson's algorithm. The match (1,_) represents a tree-search node deletion. Here we assume that an underlying evaluation function is used to guide the state expansions. Different evaluation functions prune the tree-search differently.

Inexact Algorithms. Inexact algorithms, also called error-tolerant algorithms, are used when an inexact matching is hard to find, and they use distance functions, also named cost functions, to measure the similarity between the graphs. Nilsson [140] presented an inexact subgraph matching algorithm (A*). A breath-first search on the state-space representation tree depicts the algorithm's progress. Each node in the tree-search represents a vertex in G_a that has been either matched with a vertex in G_b or deleted. If a vertex in G_a has to be deleted, it is matched to a null vertex in G_b. A cost is assigned to the matching between two vertices. The cost of a partial matching is the sum of the costs of the matched vertices. A function evaluates the partial matching by summing its cost to a lower bound estimation of the cost to match the remaining vertices in G_a. The tree search is expanded to states for which the evaluation function attains the minimum value (among all possible expansion states). The leaves of the tree (that have not been pruned) represent final states, i.e, states where all the vertices of G_a have been matched (see Figure 3.3(b)).

Llads et al. [121] applied the A^* algorithm to subgraph searching within a region adjacent graph (RAG [145]). The vertices of a RAG represent closed regions in an image and the edges represent spatial relations between different closed regions. Each vertex is associated with an attribute string containing some information about the region that it represents (e.g. the length and orientation of certain edges in the boundary contour). By using a measure of string edit operations (required to match the attribute strings), the cost function takes into account the scaling between the two regions, their shape and the structural relations with their neighborhoods.

Bunke [34] proposed a measure of similarity between two graphs based on the size of the maximum common subgraph (MCS) between two graphs. It is $1 - \|MCS(G_a, G_b)\| \ / \ max(\|G_a\|, \|G_b\|)$ and it is proved to be a metric. (We refer to [33, 21] for a describing of algorithms for finding maximum common subgraphs and their comparison). Moreover, Bunke [29] proved that the measure defined above is in linear dependence with the distance computed by an error-tolerant subgraph algorithm and more general that for some class of cost functions the inexact subgraph between G_a and G_b is equivalent of finding their maximum common subgraphs. Valiente [75] reported same results defining a measure of similarity as $\|MCS(G_a, G_b)\| - \|mcs(G_a, G_b)\|$, where mcs is the minimum common subgraph between two graphs G_a and G_b.

Approximate Algorithms. Linear programming and eigen-decomposition approaches for match graphs with the same size have been suggested in [6, 182]. Both are based on the concept of a distance function between two weighted graphs. In [6] this distance is defined as the l_1 norm of the difference between the adjacent matrix of the model graph and a permuted adjacent matrix of the query graph. In [182] the l_2 norm is used. In both cases, the matching problem is transformed to that of finding a permutation matrix that minimizes the distance between the two graphs.

Gold and Rangarajan [84] used the minimization of a cost function with respect a permutation matrix. The problem is formulated as a inequality constrained quadratic matching which is solved iteratively (graduated assignment): at each iteration an assignment problem

is used using a fast 'soft-assign' methodology.

Wilson and Hancock [194] suggested an iterative probabilistic relaxation. An initial matching probability is assigned to every pair of vertices between the two graphs. The number of all possible matchings is reduced by gradually discarding maps that contain inconsistent matching vertices. The consistency of a matching between two vertices is provided by a function defined in a Bayesian framework based on the probabilities assigned to the neighboring vertex matchings.

Wong and You [195] reported a probabilistic approach of exact and inexact subgraph matching on random graphs. Each vertex and edge of a random graph are labeled with a set of random variables. Any combination of these variables is a graph outcome of the random graph. Shannon's entropy is used to measure how many different graphs outcome from the random graph. To evaluate a matching between two random graphs the graphs are merged according an isomorphism forming a new random graph. Shannon's entropy is used to evaluate the matching. The correct matching is the isomorphism that minimizes the difference between the sum of the entropies of the input graphs and the entropy of the merge. A weighted entropy measure can be used, for inexact keygraph searching.

3.3 Keytree Searching in Tree Databases

Path-Only Searches. Many keytree searching queries are concerned only with paths [4, 28, 134]; e.g. *find the descendants of a node A who is a child of node B*. Since paths are represented as strings, existing algorithms such as AGrep [196] for string searching are applicable for processing this kind of queries (see [13, 15, 16, 81] for a review).

XISS [120] is a XML indexing and querying system designed to support regular path expressions. In each XML tree, each node is associated with a pair of integers enabling the determination of ancestor-descendant relationships among nodes. Each node in the tree is indexed according to its value and the document to which it belongs. Further, all parent-child edges are stored in an index. Processing a query consists of the following steps: (1) decompose the query into parent-child edges or ancestor-descendent paths; (2) for each

query edge, use the index to find the corresponding edges in the data trees; (3) for a query path having a Kleene star ($*$) (d1/$*$/d2), locate data node pairs (d1, d2) corresponding to the ancestor-descendent of the query path and determine whether d1 is an ancestor of d2 by using the pair of integers associated with d1 and d2; and (4) combine the results to determine whether there is a match to the query path. Additional techniques can be found in [7, 17].

Extension to Trees.When extending path-only searching to tree searching, one has to combine path matches into tree matches. Shasha et al. [162] have proposed an algorithm for searching trees named Pathfix (included in the system named ATreeGrep). Pathfix algorithm works in two phases. In the first phase, the database building phase, the algorithm encodes each root-to-leaf path of every data tree into a suffix array database [125]. In the second phase, the on-line search phase in which the query tree T_a is given, the algorithm compares T_a with each data tree T_b in the database $\mathcal{D}$ allowing a difference $DIFF$, i.e. at most $DIFF$ paths in T_a are allowed to be absent in T_b in order to consider T_b to be a match. When comparing T_a with T_b, pathfix takes every root-to-leaf path in T_a and finds roots of that path in T_b by searching in the suffix array database. (As a cutoff optimization, the algorithm stops searching T_b if more than $DIFF$ paths of T_a are missing from T_b.)

If the query tree T_a contains "don't cares", pathfix works in three steps: (1) partition T_a into connected subtrees having no "don't cares"; (2) match each of those "don't care" free subtrees with data trees in $\mathcal{D}$; (3) for the matched substructures that belong to the same data tree, determine whether they combine to match T_a based on the matching semantics of the "don't cares".

The above search process can be heuristically improved by using a hashing technique on the non-wildcard portion of data and query trees that works as follows: Compute and store all individual node labels and all parent-child label pairs in each data tree into a hash table, associating each parent-child pair with the set of data trees that contain the parent-child pair. Now suppose a query tree T_a is given with a certain distance allowed in searching, $DIFF$. Take the multiset of labels from T_a and see which data trees have a super-multiset

of those possibly with $DIFF$ missing labels. Take the multiset of parent-child pairs from T_a and see which data trees have a super-multiset of those parent-child pairs, again possibly with $DIFF$ missing pairs. This heuristic eliminates irrelevant trees from consideration at the beginning of a search and yields a set of candidate trees for further processing.

Like Pathfix, many systems [142, 14, 25, 35, 26, 45, 161, 154, 180, 200] exploit sophisticated data structures to improve the query processing time and compute the similarity between trees using a distance function. Usually the distance function is a metric[1] and all the trees in the database can be compared using a metric (i.e. the tree database represents a metric space). The systems described below work on any set of items collected in a metric space.

FQ-trees [14] organize items of a collection ranging over a metric space into the leaves of a tree data structure. Internal nodes of the tree have the same key at each level. The construction of a FQ-tree proceeds adding one by one new items to the data structure. The data structure has to be set up in advance in order to guarantee efficient retrieval. The conceptual simplicity of the technique is paid by its relative inflexibility: the internal nodes of the FQ-tree have to be designed according to a certain fixed layout of the indexes.

VP-trees [43, 200] and MVP-trees [25] organize items coming from a metric space into a binary tree. The items are stored both in leaves and in internal nodes of the tree. The items stored in internal nodes are the "vantage points". Processing a query requires the computation of its distance from some of the vantage points. The data structure is built to guarantee that "close" items are in the same sub-tree. Computation of the distance of the query from the root of a sub-tree provides useful information about the similarity of

[1] We recall that a metric d is a function that maps a pair of items O_1 and O_2 into a nonnegative number $d(O_1, O_2)$ with the following properties:

$d(O_1, O_2) \geq 0$ and $d(O_1, O_2) = 0 \Leftrightarrow O_1 = O_2$ (non negative definite) for every pair (O_1, O_2);

$d(O_1, O_2) = d(O_2, O_1)$ (symmetry) for every pair (O_1, O_2);

$d(O_1, O_3) \leq d(O_1, O_2) + d(O_2, O_3)$ (triangle inequality) for every triple (O_1, O_2, O_3).

the query with all the nodes in the sub-tree. This knowledge shows an important speed up in processing the query. More in detail, VP-trees partitions a data set according to the distances that the objects have with respect to a reference point. The median value of these distances is used as a separator to partition objects into two balanced subsets. The same procedure can be applied recursively to each of the two subsets. MVP-trees extend this idea by using multiple vantage points, and exploit pre-computed distances to reduce the number of distance computations at query time.

M-trees [45] are dynamically balanced trees. M-tree nodes store several items of the collection provided that they are ”close” and ”not too many”. If one of these conditions is violated, the node is split and a suitable sub-trees originating in the node is recursively constructed. Comparisons between M-trees and MVP-trees are reported in [25]. MVP-trees appear to have better performance than M-trees.

The method presented in this thesis inscribes itself naturally in the family of distance-based index structures searching systems. In particular, our work is an improvement of Oflazer's [142] method. We refer to Chapters 1 and 6 for an introduction and a full description of both methods,respectively. Moreover, in Chapter 6 we compare our method with the Oflazer and MVP-tree methods.

3.4 Tree matching

A simple algorithm to find the occurrences of a query tree T_a in a data tree T_b is to compare the root of T_a with each node of T_b and in the subtrees of the matching nodes recursively repeat the search for each children of the root of T_a. This takes time $O(mn)$ where n and m are the size of the compared trees. More precisely, find occurrences of query trees in data trees means solving one of the two following problems:

1. *Tree embedding*[151]. The occurrences of T_a in T_b refer to those subtrees of T_b that can be obtained from T_a by attaching new subtrees to the leaves of T_a. Here the leaves of the query tree T_a are considered as variables.

2. *Tree matching*. This is the restriction to trees of the subgraph isomorphism problem. Here we want to find where T_a appears in T_b.

Hoffman and O'Donnell [94] proposed an $O(mn)$ algorithm for the tree embedding problem. The searching algorithm is an extension of the string matching algorithm of Knuth, Morris and Pratt [109]. A query tree is represented by a set of unique paths from the root to the leaves. Each node in a path is associated with the order number of its child that is the next node on the path. The embedding is performed finding all occurrences of each query paths in the data tree.

Intense research efforts have been successful in reducing the $O(mn)$ complexity of the exact ordered tree embedding. Kosaraju [112] proposed an algorithm that takes $O(nm^{0.75}$ $logm)$ time. Dubiner, Galil and Magen [63] improved Kosaraju's algorithm by discovering periodicities in paths in the query, obtaining a bound of $O(n\sqrt{m}\ logm)$. A significant improvement, $O(nlog^3m)$, was obtained by Cole, Hariharan and Indyk [48]. Finally, other algorithms [37, 39, 178] have improved the data structures used in [94].

In general, a query tree may contain redundant nodes, removal of which would not affect answers to the query. Amer-Yahia et al [9, 10] developed algorithms for minimizing a query tree, both in the absence and in the presence of integrity constraints. Their algorithms are useful for query optimization since the size of a query tree affects the efficiency of tree pattern matching.

For the tree matching problem several method have been proposed, both for ordered [127, 124, 147, 88, 87, 185], and unordered trees [89, 90, 183]. Grossi [89] describes an $O(n + m)$ time algorithm for rooted, ordered and labeled trees, which indexes the nodes and stores all the subtrees belonging to the same equivalence class in a suffix tree. Dinitz, Itai and Rodeh [61] gave an $O(n+m)$ time algorithm for rooted, unordered and unlabelled trees. This method sorts the indices of all nodes at the same level using complex pseudo radix sorting techniques.

Special tree matching and tree embedding methods have been devised for problems which are very hard to solve exactly. To be precise we look at a new set of problems:

the approximate or inexact tree matching, and tree embedding problems. Here, the term approximate is used as synonymous of inexact; it does not refer to probabilistic or linear programming based solutions as it was in Section 3.2 for the graph matching algorithms.

Schlieder and Naumann [151] extended the exact embedding problem (problem 1) and studied the *approximate embedding* problem for unordered trees. Here, it allows a matching data tree to have nodes between a parent-child pair in the query tree, provided the ancestor-descendant relationship is preserved in the data tree. This type of embedding is also known as *tree inclusion* as defined in [104, 105, 106, 107], where Kilpelainen and Mannila showed the problem to be NP-complete. Zhang, Statman and Shasha [202] proved that approximate tree searching problem (the inexact version of problem 2) in an unordered tree is NP-complete. While linear algorithms exist for the same problem on ordered trees [176, 122, 154].

The notion of "approximation" can be further generalized by introducing a cost function that assigns a low cost value to embeddings in which none or only a small number of nodes are skipped. Schlieder and Naumann [151] presented algorithms to retrieve and rank search results using this cost function. Their algorithm is based on dynamic programming and processes the query tree T_a in a bottom up fashion. For each node q of T_a, each embedding of the query subtree rooted at q is computed from the embeddings of the query subtrees rooted at the child nodes of q. Among the valid embeddings of the query subtree $T_a(q)$ in the data subtree $T_b(q)$, the algorithm only maintains the one with the minimal cost. Repeating the above steps for each matching data node of q yields a set of embeddings of $T_a(q)$. At the end of the algorithm, the embeddings of T_a are sorted by increasing cost and presented to the user. The complexity of the algorithm is exponential though the algorithm may run much faster depending on the data.

Chapter 4

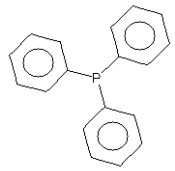
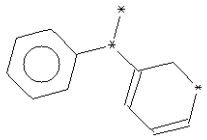
Glide: a graph linear query language

In this Chapter we present Glide (Graph LInear DEscriptor), a query language for a database of undirected graphs. The design of Glide has been influenced by two query languages: Smart [193], [97] and Xpath [46](see Table 4.1 for examples).

Smart is a query language for molecule databases coded using Smiles (Simplified Molecular Input Line Entry Specification) which is a nomenclature to represent a molecule. Smiles describes atoms and bonds of a molecule using their properties (element identity, isotope, formal charge, and implicit hydrogen count for the atoms; single, double, triple and aromatic for the bonds). Adjacent atoms are represented by concatenating the atoms specifications. If two atoms are joined by a double, triple or aromatic bond its specification is included in the representation. Smiles specifies branches by enclosing them in parentheses which may be nested or stacked. Ring closure bonds are specified by appending matching digits to the specifications of the joined atoms. Smart enriches Smiles's syntaxis including wildcards symbols to match any sequence of atoms and bonds.

XML Path Language (XPath) is a query language to address parts of an XML document. XPath models an XML document as a tree of nodes. It is based on complex path expressions where the filter and the matching conditions are included in the notation of the nodes. Adjacent node specifications are separated by the symbol "/" used in Unix file systems to describe the location of a file. Xpath specifies branches by defining the union of path expressions and it does not represented cycles. Finally, it contains wildcards to match

Table 4.1: Smiles, Smart and Xpath representations.

Graph	Smiles Representation
	C1=CC=C(C=C1)P (C2=CC=CC=C2)C3=CC=CC=C3
Query Graph	**Smart Representation**
	C1=CC=C(C=C1)*(C2=CC=C*C2)*
Query	**Xpath Representation**
Select in a XML tree elements which have 3 children	//*[count(*)=3]
Select in a XML tree (see Fig. 1.5) all elements Book and Book-Store which are children of root element DB	/DB/Book | //Book-Store

unspecified paths.

Glide uses graph expressions instead of path expressions, it represents the vertices with their labels (strings) and it uses the symbol ”/” to separate two vertices. In addition, Glide borrows the cycle notation from Smiles and generalizes it to any graph application.

4.1 Syntaxis and Semantic of Glide

The main idea in Glide is to represent a graph, in linear notation, as a set of branches where each vertex is presented only once. Vertices are represented using their labels (see Fig. 4.1 (a)) and they are separated using slashes (Fig. 4.1 (b)) ; branches are grouped using nested parentheses ’(’ and ’)’ (Fig. 4.1 (c)-(d)) and *cycles* are broken by cutting an

edge and labeling it with an integer (Fig. 4.1 (e)). The vertices of the cut edge are represented by their labels followed by %, the integer and '/'. If the same vertex is a vertex of several cut edges the label of the vertex is followed by a list of % and integers (Fig. 4.1 (g)).

Non specified components in a graph are described using wildcards '$*$','.',$+$ and ? (see Fig. 4.2). The wildcards represent single vertices or paths. The semantic of the wildcards is given in based on the elements in a graph that during a search they can match:'.' matches any single vertex; (2) '$*$' matches zero or more vertices; (3) '?'matches zero or one vertex; (3) '$+$' matches one or more vertices. Fig. 4.3 reports some examples of graphs with wildcards represented by Glide expressions.

The following definition summarizes the representations seen so far describing the grammar to construct graph expressions in Glide.

Definition 4.1.1. *A graph expression (E) in Glide is an expression generated by the following grammar:*

- $E \leftarrow \ 'label/' \mid 'label\{'\%digit'\}/' \mid './' \mid '*/' \mid '?/' \mid '+/'$
- $E \leftarrow \ EE \mid '('E')'$

where label is a string, digit a positive integer and $\{'\%digit'\}$ *is a list of one or more* $'\%digit'$.

Glide grammar allows any combination of wildcards and a recursive semantics for them, using the arithmetic of the intervals. The semantics is given in the following definition.

Definition 4.1.2. *Let* $W = \{*, ., +, ?\}$ *the set of wildcards and* U *the set of interval* $[l, u]$ *with* $l, u \in N \ and \ l \leq u$. *The function* ϕ: $W \rightarrow \ U$ *maps a wildcard to the interval of numbers of the consecutive vertices it matches* $\phi(*) = [0, \infty]$, $\phi(.) = [1, 1]$, $\phi(?) = [0, 1]$, $\phi(+) = [1, \infty]$.
The function ψ: $UXU \rightarrow \ U$ *maps combination of wildcards to the number of consecutive*

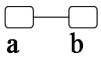
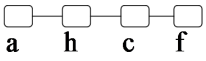
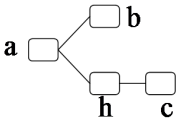
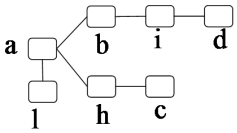
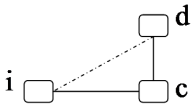
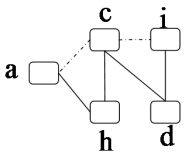
Graph	Glide Representation	Reference
	a/	(a)
	a/b/	(b)
	a/h/c/f/	(c)
	a/(h/c/)b/	(d)
	i/(b/a/(l/)h/c/)d/	(e)
	i%1/c/d%1/	(f)
	a%1/h/c%1%2/d/i%2/	(g)

Figure 4.1: Glide representation of graphs. (a) A vertex. (b) An edge. (c) A path. (d)-(e) Branches. (f)-(g) Graph with cycles. The dashed edges are the cut edges

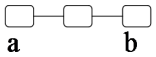
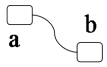
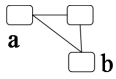
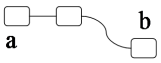
Wildcard	Semantic (matching with)	Example	
		Glide Expression	Graph
.	any (single) vertex	a/./b/	
*	zero or more vertices	a/*/b/	
?	zero or one vertex	a/?/b/	
+	one or more vertices	a/+/b/	

Figure 4.2: Wildcards in Glide.

Glide expression	Semantic
a/./././b/	Any path of length 4 beginning with a vertex 'a' and ending with a vertex 'b'
a/+/./b/	Any path of length at least 3 beginning with a vertex 'a' and ending with a vertex 'b'
a/?/./	Any path of length at least 2 and at most 3 starting with a vertex 'a'
././a/	Any path of length 2 beginning with a vertex 'a '
a%1/(*/b/)./c/d%1/	Any path where 'a' and 'c' are connected through a vertex, and a path between a and b exists.

Figure 4.3: Glide representation of graphs with wildcards.

vertices they matches

$$\psi([a,b],[c,d]) = \begin{cases} [a+c, b+d] & \textit{if } b \neq \infty \textit{ and } d \neq \infty \\ [a+c, \infty] & \textit{otherwise} \end{cases}$$

Note that in the above definition we use the symbol ∞ to mean an undefined finite number.

Glide can be used to store graphs in a compact way as follows. A depth first search algorithm can be used to generate a Glide representation of a graph. When visiting a vertex, its label is printed followed by '/' (unless this vertex is visited during a backtracking step). When the next vertex to be visited has unvisited siblings a '(' is printed. A ')' is printed when all the siblings have been visited. In a cycle, the first and last vertex representations are labeled by inserting '%' followed by a unique (for each cycle) integer before '/'.

Finally, we give some rules to construct a well formed Glide graph expressions: The minimum number of parentheses is used. For example the Glide expression "a/b/((a/b/c/) a/b/)" is not well formed. Moreover, the label of the vertices eventually concatenated with a list of '%' and integers is always followed by '/'. For example, the expressions "a/+" and "a/b" are not correct.

Chapter 5

GraphGrep: A Variable Length Path Index Approach to Searching in Graphs

In this chapter we present GraphGrep, a general method to find all the occurrences of a query graph in a database of graphs. For simplicity we focus on undirected graphs in with unlabelled edges, but GraphGrep generalizes to directed graphs with labelled edges. Due to the intractable complexity of the graph searching problem, several efficient solutions (see Chapter 3 for a review) are based on the idea to reduce the space of the possible matches. GraphGrep also utilizes such criteria. Its main algorithmic component is the storage of all paths up to a fixed length. These paths are used to perform the filtering and the matching. More precisely, GraphGrep filters out database graphs that do not contain the query graph. Moreover, for each candidate graph, it filters out the parts of the graph that do not contain the query. The matching to get an exact answer is done by combining the set of candidate paths.

In Sections 5.1, 5.2, 5.3 and 5.4 we present the three basic components of GraphGrep: (1) building the index to represent the database of graphs as sets of paths (this step is done only once), (2) filtering the database based on the submitted query and the index to reduce the search space, and (3) performing the exact matching. In Section 5.6 we report the complexity analysis of our method. Finally, in Section 5.7 we study the performance analysis of GraphGrep on random databases and we compare our method with Frowns (a

tool for searching in biochemical databases [102]) on a real molecule database [137].

5.1 Index Construction

Here we first recall some notation. GraphGrep assumes that the vertices of the data graphs have an identification number (*id-vertex*) and a label (*label-vertex*). An *id-path* of length n to be a list of n id-vertices with an edge between any two consecutive vertices (We use a Glide format to represent the label-paths). A *label-path* of length n is a list of n label-vertices. For example in Fig. 5.1(a), C/A is a label path, and (3,1) is the id-path corresponding to it. We use the label-paths and the id-paths of the graphs in a database to construct the index of the database.

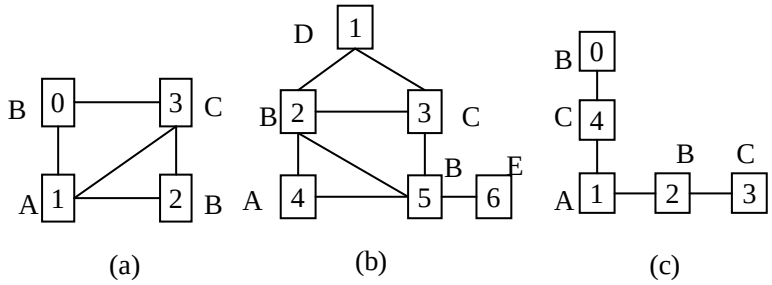


Figure 5.1: A database containing 3 graphs. (a) Graph g_1. (b) Graph g_2. (c) Graph g_3. The labels can be strings of arbitrary length.

For each graph and for each vertex, we find all paths that start at this vertex and have length one (single vertex) up to a (small, e.g. 4) constant value l_p (l_p vertices). We use the same l_p for all graphs in the database. Because several paths may contain the same label sequence, we group the id-paths associated with the same label-path in a set. The "path-representation" of a graph is the set of label-paths in the graph, where each label-path has a set of id-paths (see Table 5.1). The keys of the hash table are the hash values of the label paths. Each row contains the number of id-paths associated with a key (hash value) in each graph. We will refer to the hash table as the fingerprint of the database (see Table 5.2).

Table 5.1: Path representation of the graph in Fig. 5.1(a) with $l_p = 4$.

Label-Path	Id-Path
A/	{(1)}
A/B/	{(1, 0), (1,2)}
A/C/	{(1, 3)}
A/B/C/	{(1,0,3), (1,2,3)}
A/C/B/	{(1, 3, 0), (1,3,2)}
A/B/C/A/	{(1 ,0 ,3 ,1),(1, 2, 3, 1)}
A/B/C/B/	{(1 ,0 ,3 ,2),(1 ,2 ,3 ,0)}
A/C/B/A/	{(1 ,3 ,0 ,1),(1 ,3 ,2,1)}
B/	{(0),(2)}
B/A/	{(0,1),(2,1)}
B/C/	{(0,3), (2, 3)}
B/A/B/	{(0,1,2), (2,1,0)}
B/A/C/	{(0, 1, 3), (2, 1, 3)}
B/C/A/	{(0, 3,1), (2,3, 1)}
B/C/B/	{(0, 3, 2), (2,30)}
B/A/B/C/	{(0, 1, 2 ,3), (2 ,1, 0,3)}
B/A/C/B/	{(0 ,1, 3, 0), (2, 1, 3 ,2), (2 ,1 ,3 ,0),(0 ,1 ,3 ,2)}
B/C/B/A/	{(0 ,3 ,2 ,1),(2, 3, 0, 1)}
B/C/A/B/	{(0, 3, 1 ,0), (2 ,3 ,1 ,2), (2 ,3 ,1 ,0), (0,3,1,2)}
C/	{(3)}
C/B/	{(3,0),(3,2)}
C/A/	{(3,1)}
C/B/A/	{(3,0,1), (3,2,1)}
C/A/B/	{(3,1,0), (3,1,2)}
C/B/A/B/	{((3, 0, 1 ,2),(3 ,2 ,1 ,0)}
C/B/A/C/	{(3 ,0 ,1 ,3),(3 ,2 ,1 ,3)}
C/A/B/C/	{(3 ,1 ,0 ,3), (3 ,1 ,2 ,3)}

Table 5.2: The fingerprint of the database showing only part of rows.

Key	Graph g_1	Graph g_2	Graph g_3
h(C/A/)	1	0	1
h(C/B/)	2	2	2
h(A/B/C/A/)	2	0	0
....			
h(A/B/C/B/)	2	2	0

5.2 Decomposition of the Query

A query is a undirected labeled graph. In order to filter the database and to perform the matching, the query is decomposed in a set of intersection paths.

The branches of a depth-first traversal tree of the graph query are decomposed into sequences of overlapping label-paths, which we also call *patterns*, of length l_p or less (see Fig. 5.2).

Overlaps may occur in the following cases:

- For consecutive label-paths, the last vertex of a pattern coincides with the first vertex of the next pattern (e.g. A/B/C/B/, with $l_p = 3$, is decomposed into two patterns: A/B/C/ and C/B/).

- If a vertex has branches it is included in the first pattern of every branch (see vertex C in Fig. 5.2(c)).

- The first vertex visited in a cycle appears twice: in the beginning of the first pattern of the cycle and at the end of the last pattern of the cycle (the first and last pattern can be identical, as in Fig. 5.2(c)).

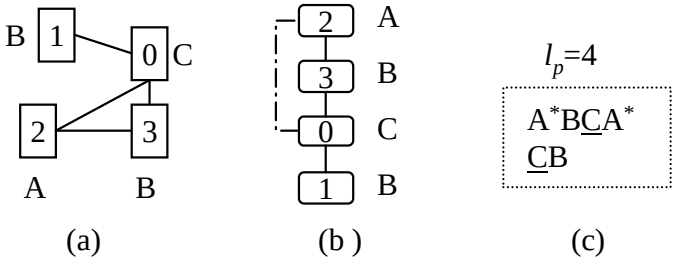


Figure 5.2: (a) A query graph. (b) The depth first tree of the graph in (a). (c) A set of patterns obtained with $l_p = 4$. In this example overlapping labels are marked with asterisks or underlining. Labels with the same mark represent the same vertex.

5.3 Filtering the Database

The query graph is parsed to build its fingerprint (hashed set of paths see Section 5.1). We filter the database by comparing the fingerprint of the query with the fingerprint of the database. A graph, for which at least one value in its fingerprint is less than the corresponding value in the fingerprint of the query, is discarded when looking for an exact subgraph match. For example, in the query graph in Fig. 5.2 with $l_p = 4$, the graphs (b) and (c) in Fig. 5.1 are filtered out because they do not contain the label-path A/B/C/A/. We call the remain graphs the candidates. The remaining graphs *may* contain one or more subgraphs matching the query. We continue filtering out parts inside of the candidate graphs in the following way: we decompose the query in patterns (see Section 5.2) and only the parts of each (candidate) graph whose label-path sets correspond to the patterns of the query are selected and then compared with the query (Section 5.4).

The above method requires finding all the label-paths up to a length lp starting from each node in the query graph. Depending on the size of the query the expected online querying time it may be an expensive job. Hence, we prefer to filter the database using the method describe above, but modifying the procedure to find fingerprint of the query graph: we use only the label-paths (patterns) found in the decomposition step (Section 5.2). The fingerprint value associated with each pattern is a single one. Obviously, this method is less

selective but it could turn out to be more efficient for some applications. In our performance tests however, we study only the unmodified method.

5.4 Finding Subgraphs Matching with Queries

After filtering, we look for all matching subgraphs in the remaining graphs. We use the path representation of the graphs to look for occurrences of the query. Only the parts of each (candidate) graph whose id-path sets correspond to the patterns of the query are selected and compared with the query. After the id-path sets are selected, we identify overlapping id-path lists and concatenate them (removing overlaps) to build a matching subgraph. For overlapping cases (1) and (2) a pair of lists is combined if the two lists contain the *same* id-vertex in the overlapping position. In overlapping case (3), a list is removed if it does not contain the same id-vertex in the overlapping positions; finally, lists are removed if equal id-vertices are not found in overlapping positions.

Example 5.4.1. Let us consider the steps to match the query in Fig. 5.2(a) with the graph g_1 in Fig. 5.1(a).

1. Select the set of paths in g_1 (Fig. 5.1) matching the patterns of the query (Fig. 5.2(c)): A/B/C/A/= $\{(1,0,3,1),\ (1,2,3,1)\}$ C/B/= $\{(3,0),(3,2)\}$.

2. Combine any list l_1 from A/B/C/A/ with any list l_2 of C/B/ if the third id-vertex in l_1 is equal to the first id-vertex of l_2 and the first id-vertex in l_1 is equal to the fourth id-vertex of l_1: A/B/C/A/ C/B/= $\{((1,0,3,1),\ (3,0)),\ ((1,0,3,1),(3,2)),$ $((1,2,3,1),(3,0)),((1,2,3,1),(3,2))\}$.

3. Remove lists from A/B/C/A/ C/B/ if they contain equal id-vertices in non-overlapping positions (the positions in each list not involved above). The two substructures in g_1 whose composition yields A/B/C/A/ C/B/ are $((1,0,3,1),(3,2))$ and $((1,2,3,1),(3,0))$.

The matching algorithm depends on the number of query graph patterns p that need to be combined; p is somewhat difficult to determine for the average case. Roughly speaking, it is directly proportional to the query size and to the maximum valence of the vertices in the query. The larger l_p, the smaller p, though this relationship is data-dependent.

5.5 Techniques for Queries with Wildcards

Query graphs with wildcards are handled by considering the parts of the query graph between wildcards as disconnected components. For example, the disconnected components of the graph in Fig. 5.3 are the path A/B/C/ and the single vertex D/.

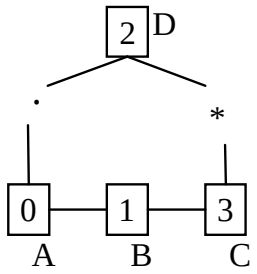


Figure 5.3: The query graph matches a graph with these properties: (1) a path between a C-labeled vertex and a D-labeled vertex may exist; (2) there is a two-edge path between an A-labeled and the D vertex; (3) there is an edge between the A vertex and a B-labeled vertex; and (4) there is an edge between the B and C vertices.

The matching algorithm described in Section 5.4 is done for each component. The Cartesian product of the sets that match each component constitute the candidate matches. An entry in the Cartesian product is a valid match if there is a path (of length equal to the wildcards' values in the query) between vertices that are connected with wildcards. The paths in the candidate graph are checked using a depth first search traversal of the graph. This step may be optimized by maintaining the transitive closure matrices of the database graphs and searching in a candidate graph only if the wildcard's value is greater than or equal to the shortest path between the vertices.

5.6 Complexity Analysis

Here is a description of the worst case complexity for GraphGrep. Let $|D|$ be the number of graphs in a database D. Let n, e and m be the number of vertices, the number

of edges and the maximum valence (degree) of the vertices in a database graph, respectively. The worst case complexity of building a path representation for the database is $\mathcal{O}(\sum_i^{|D|}(n_i m_i^{l_p}))$, whereas the memory cost is $\mathcal{O}(\sum_i^{|D|}(l_p n_i m_i^{l_p}))$. Given a query with n_q vertices, e_q edges and m_q maximum valence, finding its patterns takes $\mathcal{O}(n_q + e_q)$ time; building its fingerprint takes $\mathcal{O}(n_q m_q^{l_p})$. Filtering the database takes linear time in the size of the database. The matching algorithm depends on the number of query graph patterns p, that need to be combined; p is somewhat difficult to determine for the average case. Roughly speaking, it is directly proportional to the query size and to the maximum valence of the vertices in the query. The larger l_p, the smaller p, though this relationship is data-dependent. In general if $\tilde{n}$ is the maximum number of vertices having the same label, the worst case time complexity for the matching is $\mathcal{O}(\sum_i^{|D_f|}((\tilde{n}_i m_i^{l_p})^p))$ with $|D_f|$ the size of the database after the filtering. For a query containing w pairs of vertices connected with wildcards the complexity for the matching is $\mathcal{O}(\sum_i^{|D_f|}((\tilde{n}_i m_i^{l_p})^p + we_i))$.

5.7 Performance Analysis and Results

GraphGrep is implemented in ANSI C. The code and a demo are publicly available at `http://www.cs.nyu.edu/shasha/papers/graphgrep`. To evaluate the performance of the GraphGrep we have carried performance tests for several parameters:

- The effect of the database size in query and preprocessing time.
- The efficiency of the filtering.
- The effect of the number of the vertices of the graphs in the databases and the queries in query and preprocessing time.
- The influence of the valence (degree) of the vertices of the graphs in the database and the queries in query and preprocessing time.
- The effect of different values of l_p in the query running time and in the preprocessing time. This effect is data-dependent, but in general the larger l_p we take, the smaller

Table 5.3: Summary of notation for the different performance tests.

Symbol	Value
NDB	Number of graphs in DB
SDB	Size of database according the sum of edges of the graphs in the db plus the number of isolated vertices
SQ	Number of edges of the query plus isolated vertices
NM	Number of matches
NDBF	Number of graphs in DB after filtering

the query time and the biggest the preprocessing time.

Table 5.3 summarizes notation of the different performance tests. The experiments were performed on an Intel workstation with a Pentium 4 processor, 512 MG of memory and a 80GB disk. We use the GNU gcc on the unix-like `cygwin` layer.

The Tests Databases. We conducted numerical experiments on synthetic databases on up to 16,000 graphs; and on a database of 120,000 molecules. This database is public available from the web site of National Cancer Institute [137]. The synthetic databases are generated according the type of graphs given by Foggia, Sansone and Vento in [78]. More precisely, we create random Regular Meshes (graphs) databases; they are matrices of $n \times n$ vertices and each vertex is connect with a fixed number of vertices in its neighborhood. Meshes in 2D have vertices connected with 4 neighbors, meshes in 3D with 6 neighbors and meshes in 4D with 8 neighbors. Moreover, we consider open meshes: the nodes in the edges of the matrices have smaller degrees than the internal nodes (e.g. for 2D meshes, the nodes in the edges of the matrices are connected only with 3 neighbors). By choosing the total number of vertices *nv*, we create databases of mesh graphs with a minimum of 25 and up to *nv* vertices. Due to their regular structure they are considered a difficult case subgraph searching. We investigate the performance of GraphGrep by varying the number of vertices per graph, the number of different vertex labels, and the connectivity (2D,3D, or 4D).

Table 5.4: Preprocessing wall-clock time (in seconds) for random 2D mesh databases. The number of vertices per graph is 36 and 100.

2D	NDB	Preprocessing nv=36	Preprocessing Time nv=100
	2,000	28	76
	4,000	58	146
	8,000	117	301
	16,000	239	621

In the molecule databases, graphs (molecules) have an average number of 25 vertices; several graphs have up to 270 vertices. They are sparse graphs with maximum degree equal to 6.

The Queries. The queries are either whole graphs in databases or they are subgraphs of data graphs, generated according certain criteria that we describe bellow. The subgraphs are generated starting from a randomly selected vertex s in the graphs. We create create queries parameterizing the total number of vertices *nv* and the total number of edges *ne*. Starting from the node s we iteratively add new neighboring edges to random nodes (which are already in the query) until the termination criterion—total number of vertices or edges—is satisfied. We use Glide (see Section 4.1) to formulate wildcard queries and to specify short queries.

Results. We first study the scaling of the preprocessing step for the synthetic and molecular databases. We fix $lp = 4$. In Fig. 5.4 (a) we report the running time for molecule databases, up to 120,000 items and in Fig. 5.4 (b) we give information about the size of the graphs in the databases (which are proportional on the number of the graphs of the databases). The running time is linear in the size of the databases and this can be confirmed by the running times obtained for the random database set, Table 5.4. This linear dependence is maintained if we increase the number of vertices of the graphs in the databases, Table. 5.4, or the number of edges in the graph databases Table. 5.5.

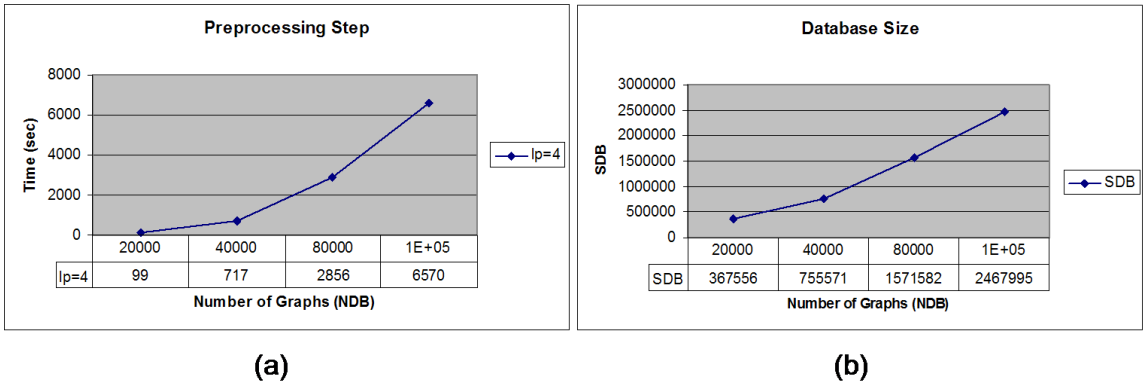


Figure 5.4: (a) The preprocessing wall-clock time (in seconds) for molecule databases. (b) The size SDB of the databases.

Table 5.5: Preprocessing wall-clock time (in seconds) for random 2D, 3D, and 4D mesh databases. The number of vertices (nv) per graph is 100.

NDB	Preprocessing Time 2D	Preprocessing Time 3D	Preprocessing Time 4D
2,000	72	348	1,236
4,000	146	735	2,678
8,000	301	1,425	5,970
16,000	621	2,987	13,750

Table 5.6: The influence of l_p in query time. Only one of the graphs in each database depicted Fig. 5.5 is returned by the filter. So we report only the number of match and the query time.

l_p	NDBF	NM	Query Time
4	1	1,024	10.34
10	1	1,024	6.5

Different values of l_p significantly effect the preprocessing step and query running time. Preprocessing (or building) the databases is more expensive with $l_p = 10$ than with $l_p = 4$; build time is exponential in l_p as we can verify in Fig. 5.5. For query Q9 in Fig. 5.8 the matching algorithm performs better when $l_p = 10$ compared with $l_p = 4$ (Table 5.6), which is consistent with the time complexity analysis.

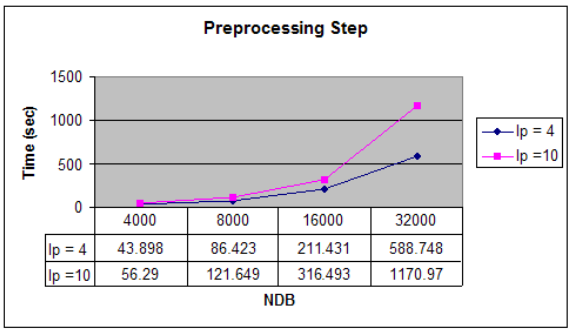


Figure 5.5: The influence of l_p in preprocessing time for a molecule database.

The running time to build the database is independent of the number of different labels in the graphs, it depends on the time to find all the id-paths of length up to l_p. On the contrary, the query running time is strongly influenced by the number of different labels in this value. Unlabelled graphs are not indexed with fingerprints. In this case a graph data is represented as a table of id-paths of length up l_p which correspond to the same label-path. The bigger the number of different labels in the graphs, the more efficiently (selective) are the fingerprints and the less matches will found. This is shown in Table 5.7, where two

Table 5.7: The influence of the number of different labels in query and preprocessing time. The query "L/L/L/L/L/L/L/" is a simple path of length 6.

NDBF	l_p	Number of Labels	Preprocessing Time	Query Time	NM
1,000	4	1	56	500	39,848,99
155	4	4	63	2.35	1,696

random databases with 1,000 graphs each are created (one has unlabelled graphs and the second has graphs with 4 different labels associated to their vertices).

The query running time. In Fig. 5.6 we report the results of querying the molecule databases, with the queries given in Table 5.8. The time to preprocess the databases is given in Fig. 5.4.

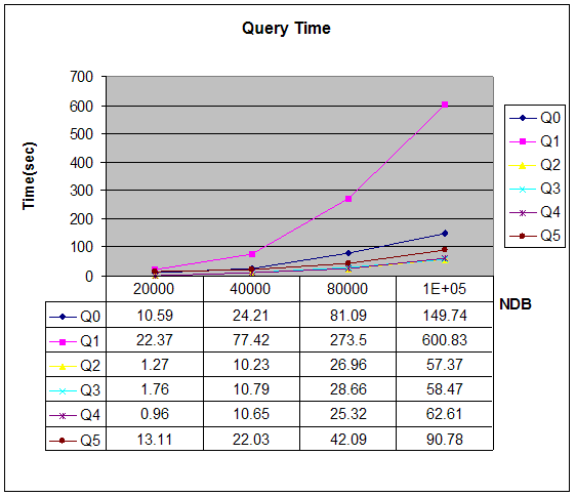
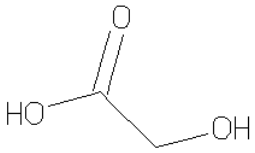
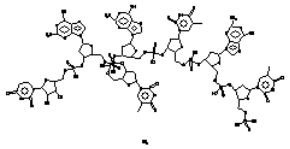
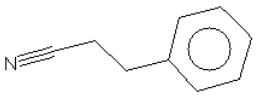
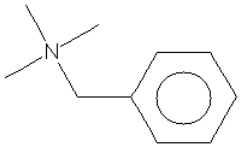
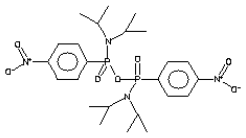
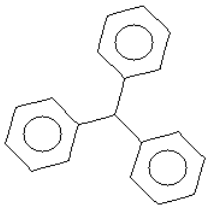
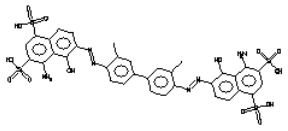
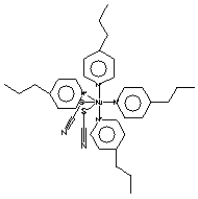
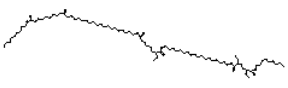
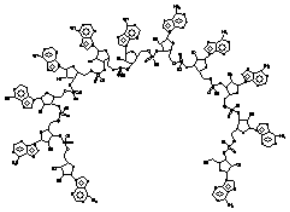


Figure 5.6: The querying time for the database of molecules.

We can observe that the querying time is proportional to the number of matches, to the query size, and to the number of filtered graphs. Matching a short query (like $Q1$) is expensive because it matches several subgraphs in many different graphs (see column NDBF and NM in Table 5.9). Bigger queries such as $Q5$ have smaller running times since they result in a smaller number of matches, and the filtering is more effective. Table 5.10 reports

Table 5.8: Molecule Queries

Query	Name	Query	Name
	Q0		Q5
	Q1		Q6
	Q2		Q7
	Q3		Q8
	Q4		Q9

the query running tine on random regular mesh graphs of type 2D, 3D, and 4D databases in Table 5.5. Here the queries are expressed using the Glide language. Tables 5.11 reports the querying time of GraphGrep on Glide wildcard queries.

Comparisons. After testing several of the parameters that influence the performance of GraphGrep we compare it with existing searching packages for molecule databases: a commercial product called Daylight [97] and an academic (open source) version of it named Frowns [102].

We do not report detailed timings of the comparison with Daylight since given a graph and a subgraph, Daylight returns the first occurrence of the subgraph, whereas GraphGrep returns all the occurrences. However, Daylight provides a web-based CGI script for such queries. We used it for verification purposes. The results for this experiment where very encouraging; both Daylight and GraphGrep returned the results within seconds.

Frowns is a free distributed searching tool for molecules database written in python. It incorporates all the features of Daylight: it works on Smiles coded databases, it generates fingerprints to filter a database and it uses the smart query language. However, different from Daylight, to perform a matching, it uses a graph-to-graph matching software, called VFLib, produced by Cordella et al. [50] (the matching algorithm is based on efficient rules to reduce the state search space of the possible mapping). Moreover, given a query it finds all the possible occurrences of the queries in the database.

To use Frowns, we generate a canonical Smiles-coded molecule database (the canonical form is used in order to take full advance of the Frowns fingerprints). For each molecule, Frowns creates its fingerprint and then it caches the fingerprint using the canonical Smiles as the key and the fingerprint object as the value into the database file. In python, most of objects can be ”serialized” and stored to disk through the shelve module. For the querying, given a molecule in Smiles query, we first convert it in canonical form and then we generate its fingerprint. The fingerprints are used to filtered out the databases. On the remains molecules we run the Frowns matching routine.

All the steps we mention above is very like what we do in GraphGrep. First we generate the fingerprint files; then we use the fingerprint to filter the database and we do the

Table 5.9: Information on querying step of Fig. 5.6.

Query	NDB	NDBF	NM
Q0	20,000	1,042	2,188
	40,000	1,945	3,734
	80,000	3,972	7,442
	12,0000	6,036	10,668
Q1	20,000	5,497	5,598
	40,000	11,825	11,146
	80,000	25,128	22,666
	120,000	41,877	40,030
Q2	20,000	1	2,048
	40,000	1	2,048
	80,000	1	2,048
	120,000	1	2,048
Q3	20,000	1	2,592
	40,000	1	2,592
	80,000	3	5,184
	120,000	3	5,184
Q4	20,000	2	4
	40,000	2	4
	80,000	7	4
	120,000	7	4
Q5	20,000	2	3072
	40,000	2	3072
	80,000	2	3072
	120,000	4	6144

Table 5.10: Information on a querying step of random a 3D meshes in a graph database with 16,000 graphs. The number of vertices in each graph is 100 and we take $l_p = 4$.

Query	NDBF	NM	Filtering Time	Matching Time	Total Time
0b%2/(0b%1/ (1b/0b/0b%1/)0b/1b%2/)0b/	56	49	0.02	0.17	0.46
0b/0b/1b/0b/0b/1b/0b/0b/	393	915	0.03	1.37	3.06
0b/0b/0b/0b/	861	6815	0.02	3.18	7.01

Table 5.11: Querying running time in GraphGrep. The queries contain wildcards. The wildcards time is spent to check if there is a path (of length equal to the wildcards' values in the query) between vertices that are connected with wildcards.

Query	NDB	NM	NDBF	FT	Match time	Wildcards time	Total Time
C%1/C/C-/*/*/C/C%1/	4,000	7,704	1,140	0.812	12.922	13.437	27.187
	8,000	2,410	18,068	1.859	43.281	30.453	75.609
	16,000	4,918	54,558	4.046	54.391	58.766	117.203
	32,000	10,278	117,242	9.89	123.421	124.61	257.921
C/(c%1/*/c%1/) (c%2/c/c/?/-c/c%2/)*/	4,000	1,814	159	0.062	1.844	1.765	3.671
	8,000	4,322	378	0.1	3.906	3.563	7.578
	16,000	11,398	1,053	0.28	10.594	9.953	20.828
	32,000	23,996	2,192	5.687	27.339	30.422	63.448

matching. The matching is performed using an algebra to combine paths remaining in the filtering step instead of using graph matching algorithms.

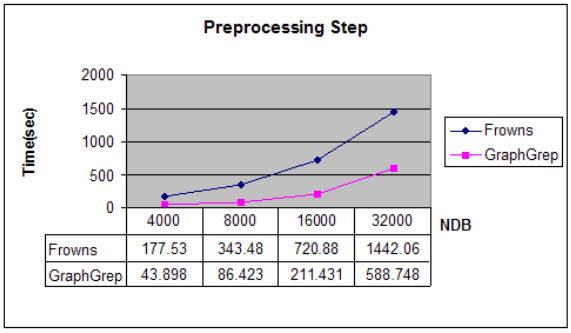
Table 5.12 reports the comparison of the querying time between Frowns and GraphGrep with $l_p = 4$ and $l_p = 10$. Here we have chosen the query Q6,Q7,Q8,Q3,Q5 and Q9 of the Table 5.8. The information related to the matching (number of matches, filtering time, matching time–excluding the filtering time, time to create the query, number of graphs in the database after the filtering) in Frowns are given in Table 5.13 and in GraphGrep are given in Tables 5.14 and 5.15 .

We note that GraphGrep performs better in all the queries, in particular for queries matching with a large number of substructures. For example, in query Q8, both systems report the same number of matches and the same number of filtered graphs. The better performance of GraphGrep is mainly due to the fact that the matching is based on the path representation of the graphs. We remark that the paths collected to build the fingerprints are stored, and used to perform the matching; this speeds up the query time but puts a premium in the required memory.

In order to achieve more accurate matches, Frowns contains optional transformations of the data to check for specific molecule properties. This transformations reduce its running time efficiency. Since GraphGrep is an application-independent system we decided to speed up the running time of Frowns avoiding usage of any transformations. In most of the cases, this did not effect the results. Sometimes, however, GraphGrep finds a larger number of matches because it does not check for rings or bonds properties. See for example query Q9 5.8 in Tables 5.13 and 5.15. Note that for $l_p = 4$ GraphGrep is slower than Frowns due to the much higher number of matches. However, with $l_p = 10$ GraphGrep is faster.

Finally, we compare the cost of the preprocessing time (see Fig. 5.7) to study the construction of the fingerprints. The efficiency of the filtering is given in Tables5.13,5.14 and 5.15. See for example the column NDBF for the query Q7. We note that overall GraphGrep is much faster in building the fingerprints and it is much more selective; the main disadvantage of GraphGrep is the memory cost. We do not report detailed results, but there is an

order of magnitude difference between GraphGrep and Frowns. We required several gigabytes of hard disk memory, where Frowns required several megabytes of hard disk memory. However, we did not try to optimize the storage format. Our fingerprints are much bigger then one used by Frowns as well the database representation. Moreover, Frowns stores the fingerprints into a database while Graphgrep stores them in its own binary format file. This is another disadvantage for GraphGrep because during querying time, given the same number of graphs in the databases after filtering, GraphGrep takes longer time to load the fingerprint file. Compare for example, column FFT in Tables 5.13 and 5.14 for query Q8, $l_p = 10$, and database size 32,000. The I/O costs of loading the fingerprint effect the performance of GraphGrep: If we run two queries consecutively (and this is what we did for all reported queries), the filtering time for the second queries will be much less because the fingerprint structures are already loaded in memory. Compare filtering times (Table 5.14) for Q6 and Q7. Both have comparable NDBFs; but because Q6 is the first query to be tested, (Q7 is next), filtering Q6 takes a much longer than filtering Q7, due to the I/O costs.



	4000	8000	16000	32000
Frowns	177.53	343.48	720.88	1442.06
GraphGrep	43.898	86.423	211.431	588.748

Figure 5.7: Comparison of Frowns and GraphGrep in Preprocessing Time

Table 5.12: The query time includes the filtering and matching time. GG stands for GraphGrep.

Query	System	4,000	80,000	16,000	32000
Q6	Frowns	6.282	14.109	40.953	98.375
	GG lp4	1.093	14.421	4.796	9.875
	GG lp10	1.109	2.156	5.406	12.953
Q7	Frowns	45.75	93.36	225.406	499.906
	GG lp4	0.39	0.843	1.453	3.578
	GG lp10	0.468	1.109	2.781	7.75
Q8	Frowns	52.656	54.36	53.344	54.735
	GG lp4	1.421	1.453	1.562	1.859
	GG lp10	1.125	1.203	1.296	6.406
Q3	Frowns	0.86	1.11	2.359	3.532
	GG lp4	0	0	0.703	1
	GG lp10	0	0	0.781	1.109
Q5	Frowns	5.204	5.437	6.078	7.171
	GG lp4	2.187	2.25	2.328	2.593
	GG lp10	2.25	2.062	2.218	2.453
Q9	Frowns	9.484	9.697	10.344	12.094
	GG lp4	10.14	11.125	9.796	10.25
	GG lp10	6.357	5.453	5.391	5.781

Table 5.13: Quering information running in Frowns. FP stands for FingerPrint, FFT stands for Filtering and Fetching Time

Query	NDB	SQ	NM	NDBF	Create FP	FFT	Matching time
Q6	4,000	11	156	137	0.063	5.812	0.407
	8,000	11	168	309	0.078	13.481	0.55
	16,000	11	360	844	0.062	39.45	1.441
	32,000	11	1488	2,000	0.047	92.527	5.801
Q7	4,000	21	1,008	1,666	0.125	40.766	4.859
	8,000	21	1,728	3,317	0.125	83.313	10
	16,000	21	4,464	7,149	0.125	199.94	25.341
	32,000	21	12,720	14,773	0.109	434.443	65.354
Q8	4,000	46	13,824	3	0.36	0.295	52.001
	8,000	46	13,824	3	0.36	0.594	53.406
	16,000	46	13,824	3	0.328	1.157	51.859
	32,000	46	13,824	3	0.36	2.313	52.062
Q3	4,000	63	0	0	0.563	0.297	0
	8,000	63	0	0	0.547	0.563	0
	16,000	63	32	2	0.516	1.311	0.532
	32,000	63	32	2	0.532	2.484	0.516
Q5	4,000	163	8	2	2.438	0.266	2.5
	8,000	163	8	2	2.406	0.594	2.437
	16,000	163	8	2	2.422	1.188	2.468
	32,000	163	8	2	2.437	2.281	2.453
Q9	4,000	271	1	1	5.938	0.296	3.25
	8,000	271	1	1	5.9	0.578	3.219
	16,000	271	1	1	5.922	1.172	3.25
	32,000	271	1	1	5.938	2.891	3.265

.

Table 5.14: Quering information running in GraphGrep. FT stands for Filtering Time

.

Query	l_p	NDB	SQ	NM	NDBF	Create Query	FT	MT
Q6	4	4,000	11	156	17	0	0.843	0.25
		8,000		168	23	0	14.015	0.406
		16000		348	71	0	4.078	0.718
		32,000		1476	209	0.016	7.718	2.141
	10	4,000		156	17	0	0.75	0.359
		8000		168	28	0	1.703	0.453
		16,000		348	80	0	3.843	1.563
		32,000		1,476	234	0	7.656	5.297
Q7	4	4,000	21	1,056	19	0	0.046	0.344
		8,000	21	1,776	40	0	0.093	0.75
		16,000	21	4,512	89	0	0.203	1.25
		32,000	21	12,864	222	0.015	0.5	3.063
	10	4,000	21	1,056	19	0	0.046	0.422
		8,000	21	1,776	40	0	0.093	1.016
		16,000	21	4,512	90	0	0.187	2.594
		32,000	21	12,864	223	0	0.484	7.266
Q8	4	4,000	46	13,824	3	0	0.078	1.343
		8,000	46	13,824	3	0	0.109	1.344
		16,000	46	13,824	3	0	0.203	1.359
		32000	46	13,824	3	0	0.5	1.359
	10	4,000	46	13,824	3	0	0.062	1.063
		8,000	46	13,824	3	0.016	0.109	1.078
		16,000	46	13,824	3	0	0.203	1.093
		32,000	46	13,824	3	0	5.218	1.188
Q3	4	4,000	63	0	0	0	0	0
		8000	63	0	0	0	0	0
		16,000	63	2,592	1	0.016	0.187	0.5
		32,000	63	2,592	1	0	0.484	0.516
	10	4,000	63	0	0	0	0	0
		8,000	63	0	0	0	0	0
		16,000	63	2,592	1	0.016	0.218	0.547
		32,000	63	2,592	1	0.016	0.484	0.609

Table 5.15: Querying information running in GraphGrep. FT stands for Fetching Time
.

Query	l_p	NDB	SQ	NM	NDBF	Create Query	FT	MT
Q5	4	4,000	163	1,536	2	0.047	0.062	2.078
		8,000	163	1,536	2	0.047	0.093	2.11
		16,000	163	1,536	2	0.063	0.218	2.047
		32,000	163	1,536	2	0.062	0.484	2.047
	10	4,000	163	1536	2	0.031	0.062	2.157
		8,000	163	1,536	2	0.031	0.109	1.922
		16,000	163	1,536	2	0.062	0.234	1.922
		32,000	163	1,536	2	0.078	0.484	1.891
Q9	4	4,000	271	1,024	1	0.157	0.046	10.14
		8,000	271	1,024	1	0.141	0.109	11.125
		16,000	271	1,024	1	0.157	0.218	9.796
		32,000	271	1,024	1	0.171	0.5	10.25
	10	4,000	271	1024	1	0.094	0.062	6.357
		8,000	271	1,024	1	0.094	0.109	5.453
		16,000	271	1,024	1	0.141	0.218	5.391
		32,000	271	1,024	1	0.141	0.515	5.781

Chapter 6

Efficient Indexing and Measurement Techniques for Error Tolerant Searching in Trees

In this Chapter we propose new methodology for fast error-tolerant retrieval of ordered trees. The new method belongs in the class of algorithms that use distance functions and indexing techniques for searching in tree databases. Our work [77] can be viewed as an improvement of an algorithm proposed by Oflazer [142]: a database of trees is organized in a compact data structure called a trie; a traversal of the trie performs the error-tolerant matching of a query tree against the database based on a distance function.

The main contributions of our work are:

- We prove the triangle inequality property for the tree-distance function introduced by Oflazer (Section 6.3).
- The triangle inequality property is used to obtain a saturation algorithm of the trie. It grants early pruning of unsuccessful branches and hence, significantly improves the efficiency of searching. (Section 6.4).
- We illustrate the effectiveness of our method with a computer vision application. When trees are used for image encoding, the proposed method can be used in pattern

recognition. In Section 6.1 we give an example of this idea, and we use it, throughout the chapter, to describe our method.

- We compare our method with the MVP-tree [25] and the Oflazer's algorithm [142]. We report results in Section 6.5.

6.1 An example of Tree Database in Computer Vision

We use a structured image database (Fig. 6.1), coded in ordered labelled tree database, to describe our method. The basic idea is to treat a structured image as a sentence in a natural language. Sentences in a natural language behave according to a general pattern. For example statements expressing the action of a subject on an object can be divided into subparts: subject, verb, object. Each subpart can be further decomposed in noun, article, attribute and so on. Naturally, this suggests the use of a hierarchical data structure, like an ordered tree, for storing and describing such sentences. The same ideas can be applied to structured images, although in this case there is more freedom (but less guidance) to choose features that better characterize each item.

The strategy adopted for the exposition and the experiments reported in this thesis depends on the semantic content of the images and requires the knowledge of an expert. This in alternative may be done in several ways, adopting different automated, semi-automated approaches or introducing sets of heuristic rules. To illustrate, in a concrete way, how the image coding is realized we report one possible scheme for some of the features of postal stamps shown in Fig. 6.1. A typical "stamp tree" can be seen in Fig. 6.2 and in Fig. 6.3.

Not all of the many possible details and features have been introduced in our scheme. This has been an intentional choice to reduce the complexity of our example and for sake of a greater clarity in exposition. Again for sake of simplicity we assume that no two distinct items will generate the same tree representation. Such a case may be avoided for example considering a larger feature set, or may be managed with slight modifications over the proposed data structure.

Figure 6.1: A database of stamps is used to illustrate our proposal referred in the text as "toy example". The following names have been used to refer to each stamp: (a) Italia650; (b) Usa60; (c) Berlin20; (d) Deutschland100; (e) Deutschland80; (f) Italia750; (g) Helvetia90; (h) Helvetia180; (i) Republique Francaise.

6.2 Indexing Construction

We proceed to explain how to store all the trees in a database in one compact data structure called trie. The trie is the base index structure used by the proposed method.

To be more precise, we adopt the "vertex list sequence" as a data structure to represent a tree T. This structure is a sequence of lists. There are as many lists in this sequence as leaves in the tree. Each list contains the ordered sequence of vertices in the unique path from the root to the corresponding leaf. For example the tree in the Fig. 6.3 is represented as follows:

((V,SMALL) (V,MORE) (V,FULL) (V,MONUMENT) (V,NORTH,LEFT,TEXT,CASTELLODI)
(V,NORTH,RIGHT,NUMBER,650) (V,SOUTH,LEFT,TEXT,ITALIA))

Using this formalism we can introduce the following definition.

Definition 6.2.1. *Let $Z = Z_1, \ldots, Z_p$, denote a generic vertex list sequence of p vertex lists. $Z[j]$ denotes the initial subsequence of Z up to and including the j-th vertex list. We*

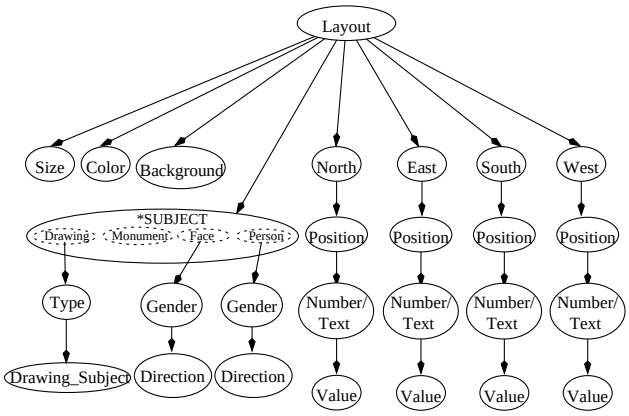


Figure 6.2: General tree of a stamp, when the semantic approach is adopted. LAYOUT $\in$ {Horizontal, Vertical, Square}; SIZE $\in$ {Small, Medium, Large}, COLOR $\in$ {1, 2, 3, More}; BACKGROUND $\in$ {Empty, Full}; TYPE $\in$ {Meaningful, Abstract}; DRAWING_SUBJECT $\in$ {Human, Vegetables, etc.}; GENDER $\in$ {Male, Female}; DIRECTION $\in$ {Left, Front, Right}; POSITION $\in$ {Left, Right, Center}; NUMBER or TEXT in each sub-region of the stamp is present with the respective VALUE. (*) As can be seen from the picture SUBJECT $\in$ {Drawing, Monument, Face, Person} and almost each category has its own subtree;

will use X (of length m) to denote the query vertex list sequence, and Y (of length j) to denote the sequence that is a (possibly partial) candidate vertex list sequence (from the database of trees).

The trees associated to the elements in our collection (Fig. 6.1) are stored in the database with their vertex list sequence representation. The set of vertex list sequences is, in turn, converted into a trie structure [76]. The trie will compress redundancies in the prefixes of the vertex list sequences to achieve a compact data structure. For instance, the trees associated with the stamps of the collection in Fig. 6.1 can be represented as a trie as shown in Fig. 6.4.

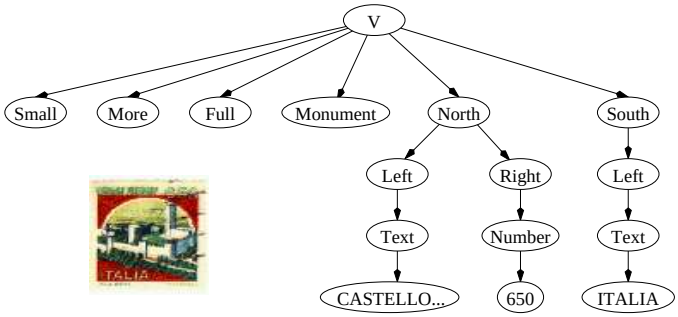


Figure 6.3: The tree of the stamp Italia650 in Fig. 6.1.

6.3 A Measurement Function between Trees

Efficient ways to retrieve in the trie all the trees ”similar”, up to some degree, to a given query have been recently reported [142]. In particular, given a query tree and a threshold value, the algorithm in [142] efficiently retrieves all the trees in the database whose distance from the input does not exceed the threshold. Observe that an objective measure to assess the quality of match is an essential element for a flexible, error tolerant, retrieval. Two trees can be different because there are two different labels in corresponding nodes or because some branch in one has no correspondence in the other (structural difference). Following [142] the distance between two trees is defined taking into account structural differences and label differences. Let C be the cost for every different label and S the cost for a structural difference. The distance between two trees, is the minimum cost of leaves or branches insertions, deletions or leaf labels changes necessary to change one tree into the other.

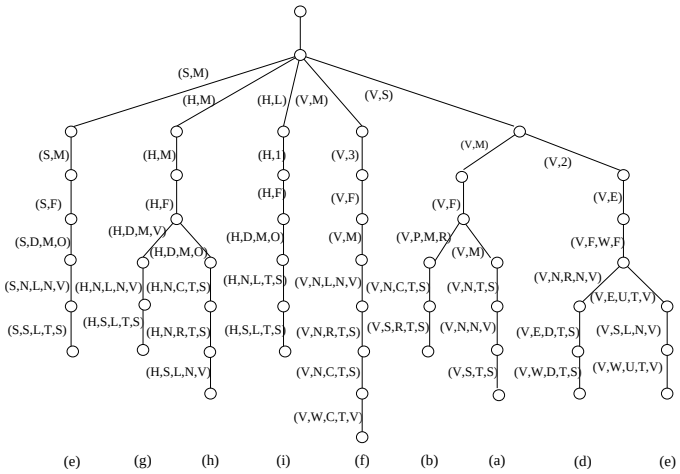


Figure 6.4: The trie associated to the collection of stamps in Fig. 6.1. The label of each edge synthesize the coding assigned by using only the first capital letter; i.e. (V,S) $\equiv$ (Vertical,Small).

More formally the distance between two trees is defined as follows (Oflazer's distance):

$$
dist(X[i], Y[j]) = \begin{cases}
dist(X[0], Y[j]) = j * S \\
\qquad\qquad \text{if } i = 0 \\
dist(X[i], Y[0]) = i * S \\
\qquad\qquad \text{if } j = 0 \\
dist(X[i-1], Y[j-1]) \\
\qquad \text{if } X_i = Y_j \text{, i. e., last} \\
\qquad \text{vertex lists are same} \\
\min \begin{cases} dist(X[i-1], Y[j-1]) + C \\ dist(X[i-1], Y[j]) + S \\ dist(X[i], Y[j-1]) + S \end{cases} \\
\qquad \text{if } X_i \text{ and } Y_j \text{ differ only} \\
\qquad \text{at the leaf label} \\
\min \begin{cases} dist(X[i-1], Y[j]) + S \\ dist(X[i], Y[j-1]) + S \end{cases} \\
\qquad\qquad \text{otherwise}
\end{cases} \tag{6.1}
$$

Table 10.1 reports the distances between all the stamps in our dimostrative collection assuming that $C = 1$ and $S = 2$. The experiments reported in Section 6.5 have been done with the same choice of values for C and S.

Table 6.1: Distances between the stamps in the database with C =1 and S = 2 when the semantic tree coding (see text) is adopted. The same code than Fig. 6.1 is adopted to identify stamps.

Dist	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
(a)	14	15	18	23	16	26	28	26
(b)		16	16	24	15	24	26	24
(c)			12	26	21	26	28	26
(d)				26	21	26	28	26
(e)					28	24	26	24
(f)						28	30	28
(g)							11	8
(h)								12

In what follows we show that the distance (6.1) is really a metric.
A metric d is a function that maps a pair of objects O_1 and O_2 into a nonnegative number $d(O_1, O_2)$ with the following properties:

M1. $d(O_1, O_2) \geq 0$ and $d(O_1, O_2) = 0 \Leftrightarrow O_1 = O_2$ (non negative definite) for every pair (O_1, O_2);

M2. $d(O_1, O_2) = d(O_2, O_1)$ (symmetry) for every pair (O_1, O_2);

M3. $d(O_1, O_3) \leq d(O_1, O_2) + d(O_2, O_3)$ (triangle inequality) for every triple (O_1, O_2, O_3).

Observe that Oflazer's distance cannot be negative because it is obtained adding up the positive costs of modifying a tree. On the other hand the image coding reported in Section 6.1 doesn't produce a canonical representation: two different images could have the same values on the features taken into account for the coding and hence may end up

producing the same tree representation. This could appear as a violation to property *M1* above. For the scope of the proposed algorithm this is not a serious problem because the search strategy is useful to match a query with a $class$ of trees that are equivalent, according to the selected features. More precisely a path from the root up to a leaf of the trie is associated to a set of trees sharing the same features values and not to a single tree. Property *M2* for the distance defined above is trivial and hence it remains only to show that the triangle inequality *M3* holds.
To do so we use the following Lemma.

Lemma 6.3.1. *Let X and Y the two vertex list sequences then if $j > i$ then*

$$dist(X[i], Y[j]) \geq (j - i) * S.$$

Proof. Proceed by induction on the pair of integer (i, j) lexicographically ordered. Consider first, $i = 0, j = 1$. Distance definition (6.1) implies $dist(X[0], Y[1]) = 1 * S = (1 - 0) * S$ and the assertion is true. Next, assume that $dist(X[i'], Y[j']) \geq (j' - i') * S$ for every $(i', j') < (i, j)$ in the the lexicographical order. Consider the three cases in the distance definition (6.1):

Case A. If $X_i = Y_j$ then $dist(X[i], Y[j]) = dist(X[i - 1], Y[j - 1])$. This, from the induction hypothesis, yields $dist(X[i], Y[j]) = dist(X[i - 1], Y[j - 1]) \geq ((j - 1) - (i - 1)) * S = (j - i) * S$.

Case B. Assume that $X_i \neq Y_j$ and X_i, Y_j differ only at the leaf label. In this case

$$dist(X[i], Y[j]) =$$

$$\min\{dist(X[i - 1], Y[j - 1]) + C,$$

$$dist(X[i - 1], Y[j]) + S,$$

$$dist(X[i], Y[j - 1]) + S\}$$

three subcases arise:

B1. If $dist(X[i], Y[j]) = dist(X[i-1], Y[j-1]) + C$ then it follows, by the induction hypothesis, that $dist(X[i-1], Y[j-1]) + C \geq (j-i) * S + C \geq (j-i) * S$.

B2. Similarly, assume

$$dist(X[i], Y[j]) = dist(X[i-1], Y[j]) + S.$$

By the induction hypothesis $dist(X[i-1], Y[j]) + S \geq (j-i+1) * S + S \geq (j-i) * S$.

B3. Finally, assume that $dist(X[i], Y[j]) = dist(X[i], Y[j-1]) + S$. The induction hypothesis yields $dist(X[i], Y[j-1]) + S \geq (j-i) * S$.

Case C. If $X_i \neq Y_j$ and X_i, Y_j differ not only at the leaf label, then two subcases identical to *B2* and *B3* arise and the very same proof of those cases may be used.

An immediate consequence of this lemma and the symmetry property *M2* of metric d is the following:

Corollary 6.3.2. *For all pairs of integers* (i, j)*,*

$$dist(X[i], Y[j]) \geq |j - i| * S$$

.

We are now ready to prove triangle inequality property of the distance function (6.1).

Theorem 6.3.3. *Let* X*,* Y*,* Z *be arbitrary list vertex sequences. Then* $dist(X[i], Z[k]) \leq dist(X[i], Y[j]) + dist(Y[j], Z[k])$.

Proof. Proceed by induction on the triple (i, j, k) of integers ordered lexicographically. First consider the case in which some of the integers i, j, k are null. We distinguish three cases:

Case A. If $i = j = k = 0$ then the theorem is trivial.

Case B. Assume that two of the indices i, j, k are null but the third one is greater than zero. Suppose for example that $i = j = 0$ and $k > 0$. Definition (6.1) yields $dist(X[i], Z[k]) = k * S, dist(X[i], Y[j]) = 0, dist(Y[j], Z[k]) = k * S$. In this case the theorem plainly holds. Similar proofs apply also to the other two symmetrical cases $i = k = 0$ and $j > 0$, $j = k = 0$ and $i > 0$.

Case C. Finally assume that only one of the indices i, j, k is null. Suppose for example $i = 0$, and j, $k \neq 0$. By definition (6.1) $dist(X[i], Z[k]) = k * S, dist(X[i], Y[j]) = j * S$. On the other hand $dist(Y[j], Z[k]) \geq |k - j| * S$ from Corollary (6.3.2). Similar proofs apply to the other two similar cases $j = 0$, and i, $k \neq 0$ and $k = 0$, and i, $j \neq 0$.

Next, assume that $dist(X[i'], Z[k']) \leq dist(X[i'], Y[j']) + dist(Y[j'], Z[k'])$ for every $(i', j', k') < (i, j, k)$ in the lexicographical order. Consider the following three cases:

A. Assume that all the three elements X_i, Y_j, Z_k are identical. By definition (6.1) it results:

$$\begin{aligned} dist(X[i], Z[k]) &= dist(X[i-1], Z[k-1]) \\ dist(X[i], Y[j]) &= dist(X[i-1], Y[j-1]) \\ dist(Y[j], Z[k]) &= dist(Y[j-1], Z[k-1]) \end{aligned}$$

The assert immediately follows by induction hypothesis.

B. Assume that only two of the elements X_i, Y_j, Z_k are identical whereas the third is distinct from the other two. For example suppose $X_i = Y_j \neq Z_k$ and Y_j, Z_k differ only

at the leaf label. By definition (6.1) it results:

$$dist(X[i], Y[j]) \;=\; dist(X[i-1], Y[j-1])$$

$$dist(X[i], Z[k]) \;=\; \min \begin{cases} dist(X[i-1], Z[k-1]) + C \\ \qquad\qquad \text{(Case a)} \\ dist(X[i-1], Z[k]) + S \\ \qquad\qquad \text{(Case b)} \\ dist(X[i], Z[k-1]) + S \\ \qquad\qquad \text{(Case c)} \end{cases}$$

$$dist(Y[j], Z[k]) \;=\; \min \begin{cases} dist(Y[j-1], Z[k-1]) + C \\ \qquad\qquad \text{(Case a')} \\ dist(Y[j-1], Z[k]) + S \\ \qquad\qquad \text{(Case b')} \\ dist(Y[j], Z[k-1]) + S \\ \qquad\qquad \text{(Case c')} \end{cases}$$

Consider all the possible combination of the case a, b, c and a', b', c'. In each of the the cases (a, a'), (b, b'), (c, c') the assert follows immediately by the induction hypothesis. Consider the cases (a, b') then

$$dist(X[i], Z[k]) \;=\; dist(X[i-1], Z[k-1]) \;+\; C,$$

$$dist(Y[j], Z[k]) \;=\; dist(Y[j-1], Z[k]) \;+\; S$$

and

$$dist(X[i], Z[k]) \;\leq\; dist(X[i-1], Z[k]) \;+\; S.$$

On the other hand by induction hypothesis $dist(X[i-1], Z[k]) \;+\; S \;\leq\; dist(X[i-1], Y[j-1]) + dist(Y[j-1], Z[k]) + S$. It follows that $dist(X[i], Z[k]) \;\leq\; dist(X[i-1], Y[j-1]) + dist(Y[j-1], Z[k]) + S \;=\; dist(X[i], Y[j]) + dist(Y[j], Z[k])$, proving

our assert in case (a, b'). Similar proofs apply to the other five subcases (a, c'), (b, a'), (b, c'), (c, a'), (c, b'). The case in which Y_j, Z_k differ not only at the leaf label is identical to the cases b, c with b', c' above. The other two symmetrical cases $X_i = Z_k \neq Y_j$, $X_i = Y_j \neq Z_k$ can be verified by similar arguments.

C. Assume that all the three elements X_i, Y_j, Z_k are pairwise distinct and differ only at the leaf label. Follows by the distance definition (6.1):

$$dist(X[i], Z[k]) = \min \begin{cases} dist(X[i-1], Z[k-1]) + C \\ \qquad \text{(Case a)} \\ dist(X[i-1], Z[k]) + S \\ \qquad \text{(Case b)} \\ dist(X[i], Z[k-1]) + S \\ \qquad \text{(Case c)} \end{cases}$$

$$dist(X[i], Y[j]) = \min \begin{cases} dist(X[i-1], Y[j-1]) + C \\ \qquad \text{(Case a')} \\ dist(X[i-1], Y[j]) + S \\ \qquad \text{(Case b')} \\ dist(X[i], Y[j-1]) + S \\ \qquad \text{(Case c')} \end{cases}$$

$$dist(Y[j], Z[k]) = \min \begin{cases} dist(Y[j-1], Z[k-1]) + C \\ \qquad \text{(Case a")} \\ dist(Y[j-1], Z[k]) + S \\ \qquad \text{(Case b")} \\ dist(Y[j], Z[k-1]) + S \\ \qquad \text{(Case c")} \end{cases}$$

Reasoning as in *Case B.* observe that in cases (a, a', a"), (b, b', b") and (c, c', c") the assert plainly follows by induction hypothesis. Suppose that the case a, b', c" holds.

Then

$$dist(X[i], Z[k]) \;=\; dist(X[i-1], Z[k-1]) \;+\; C,$$

$$dist(X[i], Y[j]) \;=\; dist(X[i-1], Y[j]) \;+\; S$$

and

$$dist(Y[j], Z[k]) \;=\; dist(Y[j], Z[k-1]) \;+\; S.$$

By applying the induction hypothesis we have $dist(X[i], Z[k]) \;=\; dist(X[i-1], Z[k-1]) + C \;\leq\; dist(X[i-1], Z[k]) + S \;\leq\; dist(X[i-1], Y[j]) + dist(Y[j], Z[k]) + S \;=\; dist(X[i], Y[j]) + dist(Y[j], Z[k])$, and our assertion follows. All the other similar cases can be treated in the same fashion. The case when some of $X_i,\; Y_j,\; Z_k$ differ not only at the leaf label is even simpler since cases a, a' and a" cannot happen.

This completes the proof of our theorem. □

6.4 Error Tolerant Searching of Trees Based on Filtering Techniques

In this section we describe several strategies to interrogate a database of trees—once it has been coded as a trie (in the way discussed above). A basic search strategy that makes use of dynamic programming is first described. Successively we report of the speed-up that is possible to attain using the triangular inequality. We conclude with a description of the complete version of the algorithm (which includes a saturation step).

6.4.1 The Basic Search Strategy

Our goal is the retrieval of trees that match a query up to some degree of approximation. Standard searching within a trie corresponds to traversing a path starting from the start node, to one of the leaves, so that the concatenation of the labels on the arcs along this path matches the input vertex list sequence. For error-tolerant searching, one has to find all paths

from the start node to leaves, such that, the corresponding vertex list sequences are within a given distance threshold t of the query vertex list sequence. To efficiently perform this search, paths in the trie that lead to no solutions have to be pruned early so that the search is bound to a very small portion of the data structure. The search proceeds, depth first, down the trie computing the similarity distance between subsequences of the query and the partial sequences obtained chaining together the labels of the nodes of the trie that have been visited in so far. Such a distance is formalized in the concept of cutoff distance. More precisely, let Q be a query, of length m, and let X_* be a candidate sequence, of a length j . The sequence X_* is the vertex list sequence obtained going from the root to a node n in the trie. Let $l = \max(1, j - \lfloor t/S \rfloor)$ and $u = \min(m, j + \lceil t/S \rceil)$ where S is the cost of an insertion or deletion and t is the threshold. The cutoff distance $cutdist(Q[m], X_*[j])$ is defined as

$$cutdist(Q[m], X_*[j]) \; = \; \min_{l \leq i \leq u} dist(Q[i], X_*[j])$$

Observe that any initial subsequence of $Q[m]$ no longer than l, requires at least $\lfloor t/S \rfloor$ leaf insertion. Similarly, any initial subsequence of $Q[m]$ no shorter than u, needs at most $\lceil t/S \rceil)$ leaf deletions. In these cases the threshold is certainly violated. The $cutdist$ function naturally suggests a recursive dynamic programming approach. The leaves that have been reached during the search, within the cutoff distance bounds are the output of the error tolerant matching procedure. Oflazer [142] has shown that this kind of search can be realized in O($L^2 log L k^{1/L\lceil t/S \rceil}$) where L is the number of leaves in each tree, k is number of the tree in the forest, t is the threshold distance and S is the cost of adding or deleting a leaf in a tree. A pseudo code description of the procedure described above is provided in Fig. 6.11.

6.4.2 Search Strategy Based on Triangle Inequality Property

The are several ways to optimize the basic search strategy. The first one is to use of the triangle inequality property for Oflazer's distance (Theorem 6.3.3). For every triplet of

objects (x, y, q), triangle property implies that:

$$|dist(x, q) - dist(x, y)| \leq dist(y, q) \tag{6.2}$$

Two cases may occur ([191]):

1. If $dist(x, q) \geq t$, it is possible to discard x. If there is another object y close to x such that $dist(x, q) - dist(x, y) \geq t$ then, by the inequality (6.2), it is possible to deduce that $dist(y, q) \geq t$. y may hence be discarded without having to calculate $dist(y, q)$. Fig. 6.5 (a) illustrates this case.

2. If $dist(x, q) \leq t$ then x matches the query within the prescribed error bound. If y is very far from x in such way that $dist(x, y) - dist(x, q) \geq t$ then, by the triangle inequality, we can deduce that $dist(y, q) \geq t$. Therefore as in the previous case we are entitled to discard the object y without having to calculate $dist(y, q)$. This case is illustrated in Fig. 6.5 (b).

Figure 6.5: The two diagrams illustrate the two applications of triangle inequality discussed in the text.

The above applications of triangle inequality allow an improved search of the trie. To better explain the improved strategy we introduce the following definitions:

Definition 6.4.1. *Given a tree, a node k is a $keynode$ if*

1. *Every node in the path from k to a leaf has out-degree at most 1;*

2. *the parent of k does not have property 1.*

Therefore, in a trie structure a key node is the highest node such that it uniquely defines a tree of the forest corresponding to only one object of data collection. Fig. 6.6 illustrates this concept.

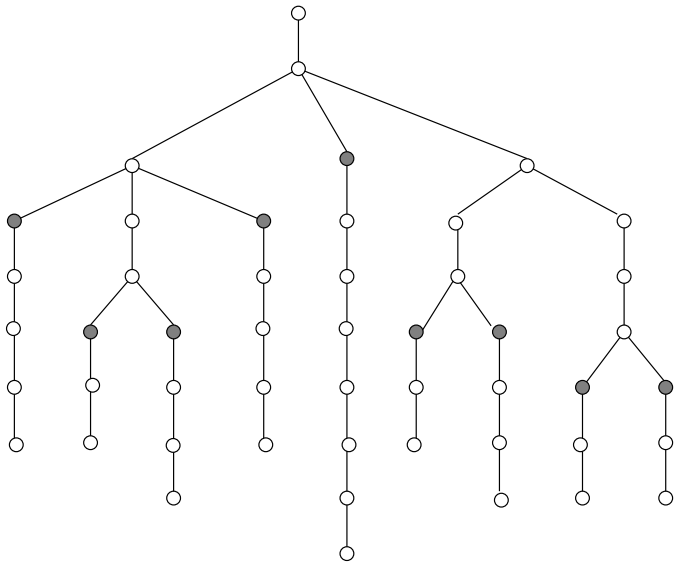


Figure 6.6: In the picture the darker nodes are the $keynodes$ of the trie.

The improved search algorithm proceeds as follows: a depth first search of the trie is performed starting from the root of the trie until one of the two following cases arise:

Case 1. When the visit reaches a node n the similarity threshold t is exceeded: the trie is hence pruned at node n and the search backtracks along other paths. The pruning may be also performed on the family of all the key nodes of the trie that are ancestors of leaves Y_k's satisfying the following condition:

$$cutdist(Q[m], X_*[j]) - dist(Y_k[l_k], X_i[l_i]) \geq t \tag{6.3}$$

for some leaf X_i of the sub-trie with root n. In the above condition $X_*[j]$ indicates the subsequence (of length j) from the root of the trie to the node n; l_k and l_i indicate the lengths of the sequences Y_k and X_i, respectively. Of course the extra

pruning is of interest only if the leaf Y_k has not been yet pruned or visited. The rationale behind this extra pruning is in the observation that $cutdist(Q[m], X_*[j])$ is a partial computation of $dist(Q[m], X_i[l_i])$ and $dist(Q[m], X_i[l_i]) - dist(X_i[l_i], Y_j[l_k]) \geq cutdist(Q[m], X_*[j]) - dist(X_i[l_i], Y_j[l_k]) \geq t$. See Fig. 6.7.

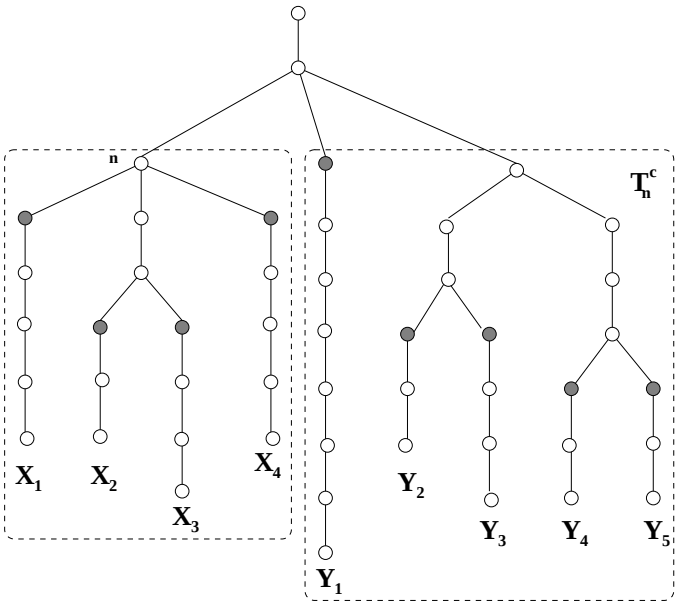


Figure 6.7: The figure illustrates the situation when *Case 1* occurs: the dissimilarity between node n and the query exceeds threshold t. The sub-trie rooted at n is pruned. All the sub-tries rooted at key nodes inside the dashed rectangle on the right of the picture denoted with $\mathbf{T_n^c}$, may be pruned provided that their descendant leaf Y_k satisfies condition (6.3).

Case 2. A leaf x is reached within the similarity threshold: the corresponding object X is part of the output set. Even in this case, a pruning of unvisited portions of the trie may be performed. More precisely, all unvisited key nodes whose descendent leaf Y_k satisfies the condition $dist(X, Y_k[l_k]) - dist(X, Q[m]) \geq t$ are pruned (see Fig. 6.8).

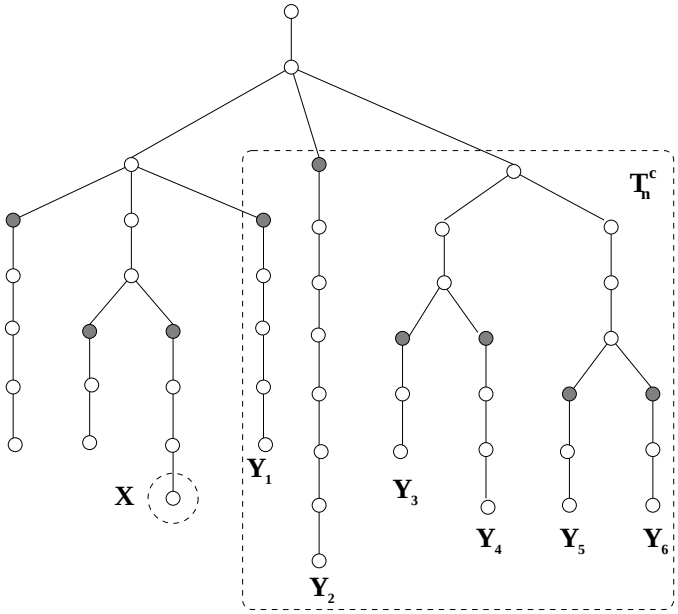


Figure 6.8: The figure illustrates the situation when *Case 2* occurs: the dissimilarity between leaf X and the query does not exceed threshold t. All the sub-tries rooted at key nodes inside the dashed rectangle on the right of the picture denoted with $\mathbf{T_n^c}$, may be pruned provided that their descendant leaf Y_k satisfies condition $dist(X, Y_k[l_k]) - dist(X, Q[m]) \geq t$.

6.4.3 Search Strategy Using Saturation

A further improvement of the search strategy can be obtained by adopting a saturation technique. The saturation step takes place after the pruning step done using triangle inequality (see Fig. 6.9 and Fig. 6.9). More precisely when the pruning described above is performed, we ”mark” a field of the node key where the cut is done. To take advantage of such a marking and hence to further reduce the search space, the marking has to be “propagated” upward in the trie. In this way entire branches of the trie will not be visited in the successive search phase. This propagation is performed as follows: do a post-order visit of the trie and ”mark” in the same way all the nodes such that all their children are already ”marked”. The effect of the application of the procedure is illustrated in Fig. 6.10.

```
SATURATION(n)

1  if n is not a key node and n is not marked then
2    for each child f of n do
3              SATURATION(f);
4    endfor
5    if all children of n are marked then
6      mark n;
7  return;
```

Figure 6.9: The Saturation algorithm. The procedure marks all nodes that have all their children already marked.

We summarize all optimization techniques proposed above in a final description of the procedure reported in Fig. 6.12. The original Oflazer's algorithm is reported, for comparison, in Fig. 6.11.

6.5 Performance Analysis and Results

The evaluation of a search strategy for large databases is a complex task. One has to take into account both theoretical and practical issues. Experiments in this respect are an invaluable tool to assess properties and limitations of any given technique. In this section we report the results of several tests performed on a moderate size database. In particular we used databases of postal stamps of different sizes from 100 up to 300 items.

The database has been obtained with the help of a human expert. The expert's intervention provided two actions: first to help in choosing a suitable tree structure to store relevant features of each item in the collection; second to obtain the actual trees describing the items. Observe that this step could be, in principle, completely automated provided that good heuristic feature extraction techniques are available.

It is preferable to obtain estimates about efficiency regardless of a particular architecture and regardless to implementation details. In search of objective measures that could be at

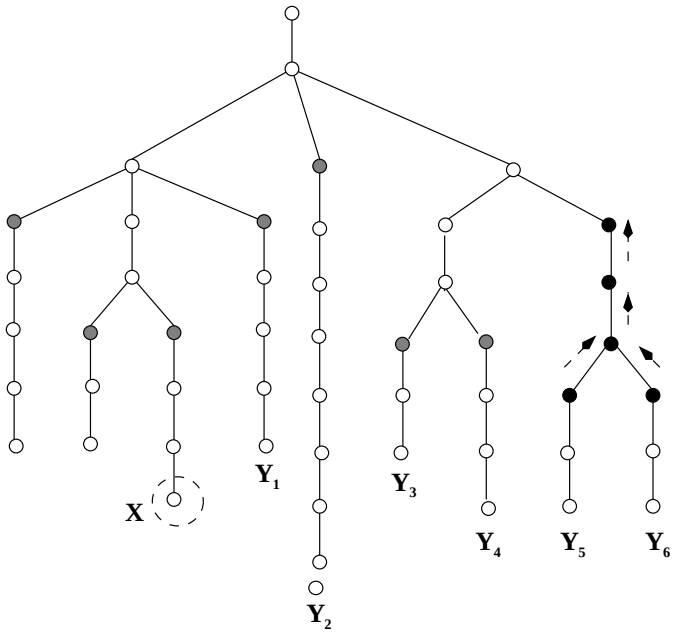


Figure 6.10: The figure illustrates the effects of the saturation on the trie in Fig. 6.8. The key nodes ancestor of Y_5 and Y_6 have been marked as an application of triangle inequality. They are shown in black as well as all the other nodes that are "marked" in the saturation procedure.

the same time simple to get and expressive of the computational resources required by a search strategy we have identified the following performance indicators:

- the average execution time per query;
- the number of distance calculation per query.

The first measure takes into account all the computation resources needed to manage the data structures and the details about data management. One can roughly say that average execution time per query shows a direct correlation with the complexity of the items in the database. It is hence needed to pair this performance measure with second indicator that is independent from the particular nature of the items in the database. The second measure proposed above, the number of distance calculation per query, provides an indicator

OFLAZER_SEARCH

```
1  Push((ε, q_0));
2  while stack not empty do
3        Pop((X'_*, q_a));
4        for all q_b and V |
         δ(q_a, V) = q_b do
5            X_* := concat(X'_*, V);
6            if cutdist(Q[m], X_*[j]) ≤ t then
7              Push((X_*, q_b))
8            if dist(Q[m], X_*[j]) ≤ t and
             q_b is terminal node then
9              Output X_*;
```

Figure 6.11: Oflazer's algorithm for approximate tree retrieval. t denotes the similarity threshold, ϵ the empty vertex list sequence, q_0 the root of the trie, q_s a node of the trie, Q the query with length m, X_* the partial vertex list sequence under consideration with length j, V the label of a edge in the trie denoting a vertex list and $\delta(q_i, V)$ the node of the trie which can be reached from node q_i by V.

as requested above.

We have also observed that in the approach discussed in this thesis the proportion of computation resources required by any query is mostly made by the cost of distance computation.

The proposed approach requires a preprocessing phase to compute distances between objects in the database. The exact number of cycles used time by this step depends of course, on the architecture of the processor. A quadratic growth of the number of CPU cycles as the number of the items in the database increases has been observed in perfect agreement with the theoretical prediction.

The order of magnitude of the preprocessing time, is about 10^3 with respect to the average processing time required by a single query. On the other hand the preprocessing may be done off line only once. These considerations imply that the proposed approach is best suited when many queries have to be performed on the database without requiring upgrades

IMPROVED_OFLAZER_SEARCH

```
1  Push((ε, q0));
2  while stack not empty do
3        Pop((X'*, qa));
4        for all qb and V |
         δ(qa, V) = qb and mark(qb) = FALSE do
5            X* := concat(X'*, V);
6            if cutdist(Q[m], X*[j]) ≤ t then
7              Push((X*, qb))
8            else
9              mark(qb) := TRUE;
10             PRUNING-CASE-1(qb);
11             SATURATION(q0);
12           if (dist(Q[m], X*[j]) ≤ t and
             qb is terminal node) then
13             Output X*;
14             PRUNING-CASE-2(qb);
15             SATURATION(q0);
```

Figure 6.12: Improved Oflazer Algorithm using triangle inequality property and saturation procedure. The same notation of Fig. 6.11 is used. Moreover, $mark(q_s)$ set to true indicates that the node q_s has been pruned, the function SATURATION is summarized in Fig. 6.9, the functions PRUNING-CASE-1 and PRUNING-CASE-2 implement the techniques described in Section IV .B under the *Case 1* and *Case 2*, respectively.

of it. Note that the triangular property is particularly useful when it is applied on triple of objects that are far apart in the trie: accordingly good speed up factor have been observed in our experiments even if the computation of mutual distances is done only for a small subset of all the pairs of objects in the database.

In the experiments the pairwise distances of all the objects in the database range over the interval [0,36]. Moreover over such a range the distances are roughly uniformly distributed. The performance assessment is made measuring the average processing time per query and the average number of distance computation per query over databases of different sizes as

the similarity threshold goes from a minimum to a maximal value. More precisely we have performed experiments with collections of 100, 150, 200, 250, 300 stamps and thresholds 2, 4, 6, 8, 10. The average query processing times are diagrammed in Fig. 6.13 (b). In Fig. 6.13 (a) It is possible to compare the observed performance indicators to the same indicators obtained adopting the not optimized version of the algorithm. The experiments

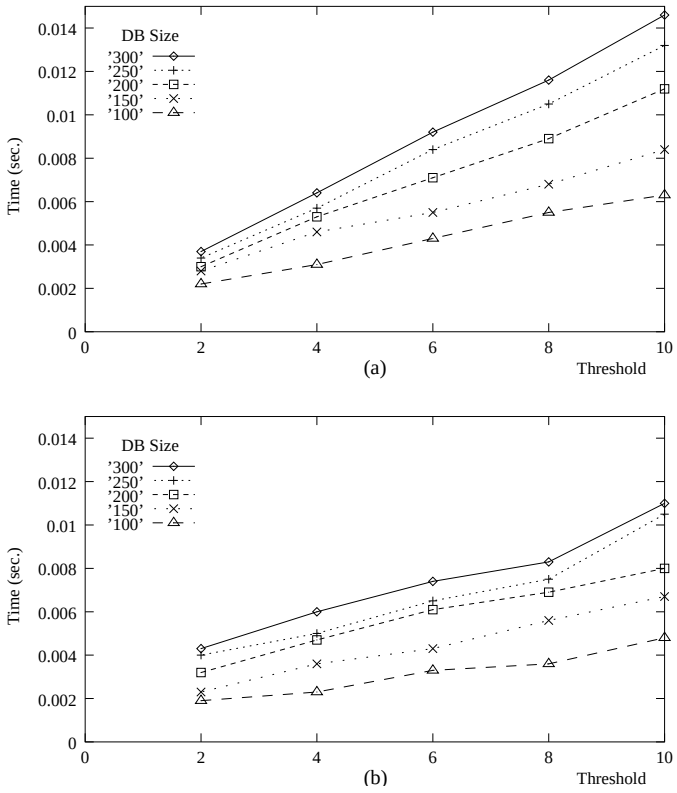


Figure 6.13: Observed average execution time per query for databases of different sizes, for different threshold values; (a) Oflazer's original algorithm; (b) proposed algorithm.

show that the improvement becomes relevant as the size of the database grows. The choice of a suitable threshold value is influential. In order to better illustrate its role, we adopted, in Fig. 6.14, a different visualization method. The picture shows the growth of average query processing time as the threshold ranges over [4,10]. As for the average number of calls to the $cutdist$ function we report the observed values in Fig. 6.15 and Fig. 6.16, according

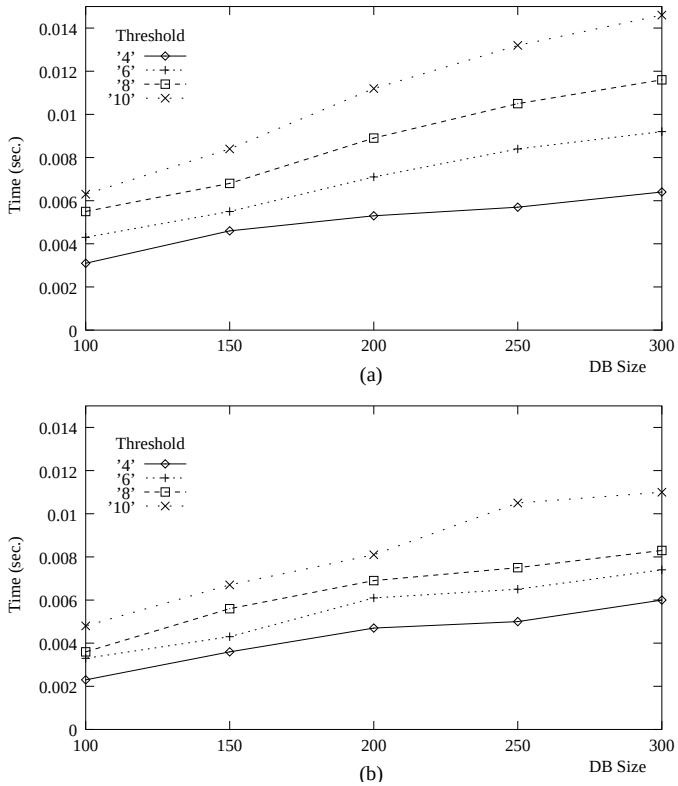


Figure 6.14: Observed average execution time per query for different threshold values, for databases of different sizes; (a) Oflazer's original algorithm; (b) proposed algorithm.

to the same visualization methods adopted in Fig. 6.13 and Fig. 6.14. In order to assess the quality of performance of the proposed algorithm we compare it with other search strategies based on distance computation. Candidates for comparisons are the following search strategies that have been discussed in the introduction and that are all based on distance computation:

- FQ-trees ([14]);
- MVP-trees ([25], [43], [200]);
- M-trees ([45]).

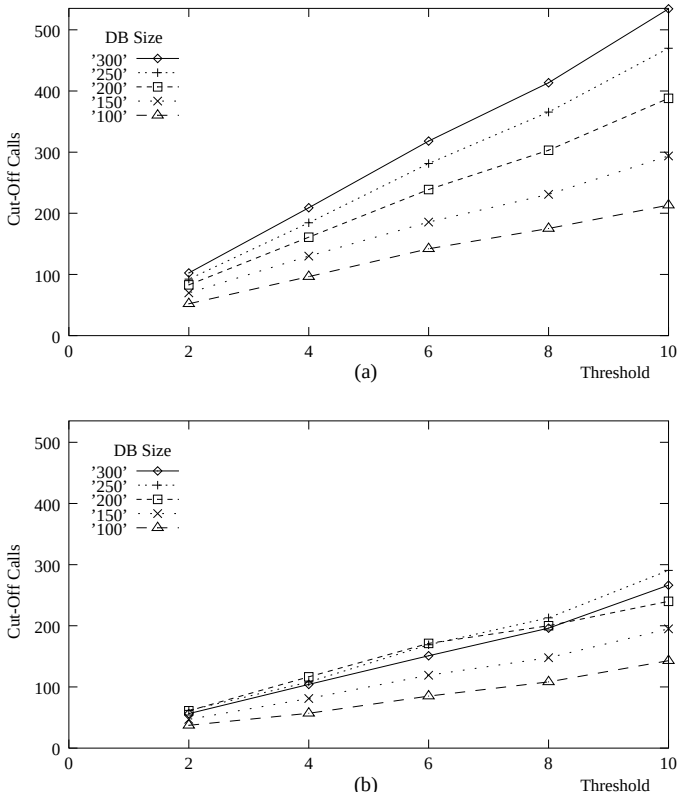


Figure 6.15: Observed average number of calls to function $cutdist$ per query for databases of different sizes, for different threshold values; (a) Oflazer's original algorithm; (b) proposed algorithm.

For these methods, as well as the proposed one, the theoretical complexity of query processing is, in the worst case, linear in the size of the database.

Hence, to get meaningful information about the different techniques is important to consider the number of calls done to a distance function.

We choose not to compare our algorithm directly with FQ-trees because the performance of FQ-trees is close to the performance that can be obtained with the proposed approach, provided that a suitable indexing layout for the data structure is made. Our technique, however, allows greater flexibility and simplicity in setting up the right data structure. Several

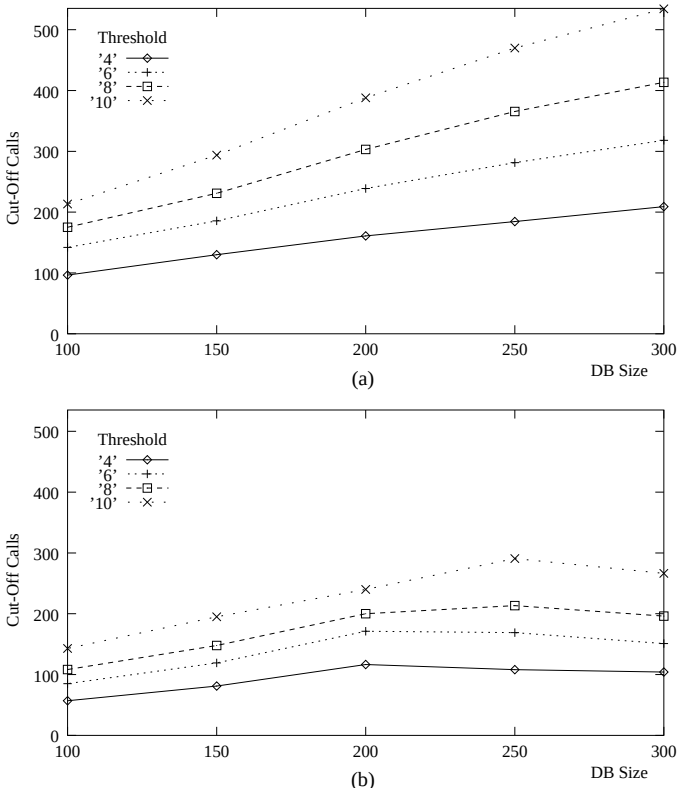


Figure 6.16: Observed average number of calls to function $cutdist$ per query for different threshold values, for databases of different sizes; (a) Oflazer's original algorithm; (b) proposed algorithm.

comparisons between MVP-trees and M-trees are available in the literature. More precisely, [25] proposes a direct comparison of MVP-trees with M-trees.

For the reasons discussed before we choose to compare our technique only with the MVP-trees. We have implemented MVP-trees homogeneously with the implementation of the proposed method and we have recorded the average number of calls per query to the distance function. The results of the experiments are summarized in Fig. 6.17 and clearly show that the proposed technique works better than MVP-trees relatively to this performance indicator.

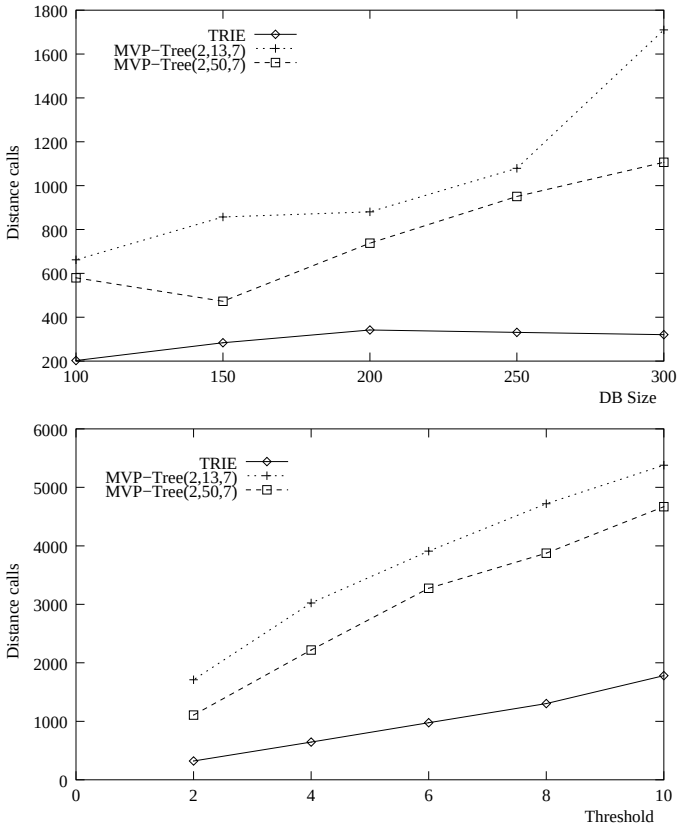


Figure 6.17: Number of distance function calls as function of the threshold and of database size. The same notation used in [25] is adopted hence $MVP - Tree(m, k, p)$ represents the MVP-tree with m partitions created by each vantage point, with fan out k for the leaf-nodes, with p distances for the data points to be kept at the leaves. Upper diagram shows the number of calls to distance function as the database size increases with a fixed threshold = 6; lower diagram shows the number of calls to distance function as the threshold increases with a database of fixed size = 300.

Chapter 7

Conclusions

In this part of the thesis we have focused on pattern-matching based algorithms for fast searching in trees and graphs.

We first presented a search algorithm GraphGrep and a query language Glide for database of graphs. GraphGrep is an application-independent method finding all the occurrences of a query graph in a databases of graphs. This is an important problem in many applications from computer vision to chemistry. Several searching systems exist for chemical databases but no much research has been done for application-independent searching in database of graphs. Our research will therefore permit inexpensive data analysis across many domains.

The algorithms involve a relational and string searching sublanguage on an algebra. The primitives involve an indexing technique which represent the database graphs as a collection of small subgraphs (typically paths containing also cycles), joins between paths and selections of nodes. The above index is used in query time to filter out whole graphs or parts of graphs which will not match with the query. This allows to GraphGrep to be very efficiently for applications searching for small graphs in database of graphs. The net result is a complexity with no exponential dependency on the data graph, linear in the size of the database and exponential in the size of the query graph.

Our software has been tested on random databases and molecule databases. We have compared GraphGrep with the best known software, (Daylight and Frowns) and we have

obtain very promising results.

GraphGrep accept query in two following format: as list of nodes and edges or using a new query language named Glide. Glide is a useful combination of Xpath with Smile. Xpath is an XML standard and Smiles is a biochemical standard. By combining the two, Glide allows the expression of a wide variety of queries including queries having fixed and variable length don't cares. Finally, GraphGrep is implemented in C, the code and a demo are available at http://www.cs.nyu.edu/shasha/papers/graphgrep/.

Concerning the searching in tree databases, in this part of thesis we have presented an algorithm for an approximate matching of subtrees in a database of trees. Our methodology is based on the retrieval technique introduced by Oflazer [142] and it is improved by the application of the triangle property in connection with a saturation procedure applied to the trie used to store the images. The results show that the techniques is valuable for the construction of large databases of visual information and other similar objects. The experiments also show that the improvement due to proper usage of triangle property in connection with the saturation algorithm is relevant both in terms of number of distance calculations and of execution time. The improvement becomes more significant when the dimension of the database and the complexity of the objects increase. Finally, the performance of the proposed technique in comparisons with other distance based search methodologies is very promising.

Future work on searching in tree and graph databases field includes:

- Improve the performance of GraphGrep so that it can be as fast as keyword searching engines like google. It could be embarassingly parallelizable so will scale well.
- Develop indexes that trade time for space optimally (storing all paths may be more than is needed, but storing just parent-child pairs may be too little).
- Develop practically meaningful distance measures on graphs and approximate query processing algorithms to support inexact matching.
- Develop a framework for selectivity estimation for queries on trees and graphs with

wildcards.

- Develop a framework for turning searching to pattern discovery in trees and graphs [56, 188, 189, 192].
- Develop support for semantic extensions: semi-flexible or flexible queries [100] in which parent-child relationships in queries may become ancestor-descendant or even descendant-ancestor relationships in data graphs.

Part II

Probabilistic, Temporal and Objects Database Searching

Chapter 8

State of Art

Several researchers have worked on object oriented databases that support temporal features or uncertainty features but to our knowledge there is no previous work in integrating and combining these two features in the same object bases.

On the contrary, for *relational* databases integration of time and probability has been proposed. Dyreson and Snodgrass [67] extend the *SQL data model and query language* by probabilistic uncertainty on time points. They add indeterminate temporal attributes (which have indeterminate instants as associated values) to SQL. Indeterminate instants are intervals of time points with associated probability distributions. The SQL query language is extended by a construct to define *ordering plausibility* which is an integer between 1 and 100 that specifies the degree to which the result of an SQL query should contain uncertain answers (1 means that any possible answer to a query is desired, while 100 says that only definite answers to a query are desired). The *correlation credibility* construct specifies simple modifications of probability distributions in the base relations before evaluating the selection condition in SQL queries. Dyreson and Snodgrass also describe efficient data structures and query processing algorithms for their approach. Dekhtyar et al. [57] extend the *relational data model and algebra* by temporal indeterminacy based on probabilities. They define a *theoretical annotated temporal algebra* on large *annotated relations*, and a *temporal probabilistic algebra* on succinct *temporal probabilistic relations*. They show that the latter efficiently and correctly implements the former. They also report on timings

of the temporal probabilistic algebra in a prototype implementation.

Our work is closest in spirit to the above work by Dekhtyar et al. [57]. The idea of having an explicit algebra on large instances, which is efficiently and correctly implemented by an implicit algebra on succinct instances, is inspired by their work. Our work is an extension of the much richer object-oriented data model and algebra, as compared to the relational algebra. Thus, our work may be viewed as a generalization of Dekhtyar et al. [57].

Dyreson and Snodgrass's work [67] differs from ours in several ways. First, we present an extension of object-oriented databases, while their approach is an extension of relational databases. Second, we make no independence assumptions between events (the user's query can explicitly encode her knowledge of the dependencies between events, if any), while Dyreson and Snodgrass assume that all indeterminate events are probabilistically independent from each other. Third, our work introduces an algebra, while their work defines an SQL extension. Fourth, we present formal definitions of important notions like coherence and consistency and show that under appropriate assumptions, our operations all preserve coherence and consistency. Fifth, we allow for interval probabilities over solution sets of temporal constraints, while their work allows only for precise point probabilities over intervals of time points.

Our work is also related to data models and algebraic operations for complex objects [2, 165, 184, 175, 172, 22]. Our work is a strict extension of the algebra for complex values presented by Abiteboul et al. [2]. As in the case of Shaw and Zdonik [165], Vandenberg and DeWitt [184], and Boncz et al. [22], our data model supports the type constructors for sets and tuples on elementary datatypes. Like them, we also support the algebraic operations of selection, projection, join, union, intersection, and difference. However, unlike them, we do not support the type constructors for arrays and multisets as in [184], user-defined abstract datatypes as in [165], and classes as elementary datatypes as in [22], and aggregate operations and next/unnest operations. Of course, our work involves time and probabilities that are not considered in these papers. Extending our work to such types and algebraic operations is an interesting topic of future research. The nested relational

algebra described in [175] is a functional language for complex objects, which also allows for defining the high-level algebraic operations of selection, projection, Cartesian product, intersection, and difference [175]. Finally, Subramanian et al. [172] describe an object-oriented query algebra for lists and trees. They also present a predicate language for lists and trees, which supports order-sensitive queries, as it is based on pattern matching. Such algebraic operations are in some sense related to our extraction operation, which extracts a subhierarchy from the class hierarchy of a temporal object base.

Chapter 9

Types

In this

In this Chapter we define the types manipulated by a temporal probabilistic object database and we recapitulate some basic notions in probability theory. We first recall the notion of a calendar due to Kraus et al. [115], which serves as a temporal atomic type in our model. We then define types and their values. The set of all values of a type τ is also called the *domain* of τ, denoted $\mathrm{dom}(\tau)$. We first introduce (non-probabilistic) classical types and their values. We then define probabilistic types and their values. Finally, we describe the concept of a probabilistic strategy, which is used to combine probabilistic information in our algebraic operations in Sections 11 and 12.

9.1 Calendars

Intuitively, a calendar consists of a finite sequence of time units and a predicate specifying a set of valid time points over this sequence. It is used as an elementary temporal type with the set of all valid time points as associated domain of values.

We first define time units and linear temporal hierarchies, which are essentially finite sequences of time units. A *time unit* $T = (N, V)$ consists of a *name* N and a set of *time values* V. We often use N to refer to T. A *linear temporal hierarchy* $H = T_1 \sqsupseteq \ldots \sqsupseteq T_n$ consists of a finite set of distinct time units $\{T_1, \ldots, T_n\}$ and a linear order $\sqsupseteq$ among them.

The following example illustrates the above concepts.

Example 9.1.1. $T_y = (year, \{0, 1, \ldots\})$, $T_m = (month, \{1, \ldots, 12\})$, and $T_d = (day, \{1, \ldots, 31\})$ are time units, while $T_y \sqsupseteq T_m \sqsupseteq T_d$, or $year \sqsupseteq month \sqsupseteq day$, is a linear temporal hierarchy. □

A linear temporal hierarchy specifies a set of time points, while a calendar additionally specifies a subset of valid time points. More formally, a *time point* over $H = T_1 \sqsupseteq \ldots \sqsupseteq T_n$ is a tuple $(t_1, \ldots, t_n)$, where each t_i is a time value of T_i. We denote by $<_H$ the lexicographic order on all time points over H, which is defined by: $(s_1, \ldots, s_n) <_H (t_1, \ldots, t_n)$ iff some $i \in \{1, \ldots, n\}$ exists such that $s_j = t_j$ for all $j \in \{1, \ldots, i-1\}$ and $s_i < t_i$. We use $\leq_H$ to denote the reflexive closure of $<_H$. A *calendar* $C = (H, P)$ consists of a linear temporal hierarchy H and a *validity predicate* P, which specifies a nonempty set of *valid* time points over H. A calendar is *finite* if the set of all its valid time points is finite. In the rest of this paper, all calendars are finite unless specified otherwise. The reader interested in how to specify validity predicates may consult [115]. We give an example to illustrate the concepts of time points and calendars.

Example 9.1.2. $(1997, 1, 31)$ and $(1997, 2, 31)$ are time points over $H = year \sqsupseteq month \sqsupseteq day$. In a calendar $C = (H, P)$, the validity predicate P may now characterize the former as valid and the latter as invalid. □

9.2 Classical Types

A calendar τ is a *temporal atomic type* whose domain dom(τ) consists of all the valid time points of τ. The set of *classical atomic types* is $\mathcal{T} = \{\mathsf{integer}, \mathsf{string}, \mathsf{Boolean}, \mathsf{float}\}$ with the usual domains. We also assume the existence of some arbitrary but fixed set $\mathcal{A}$ of attributes, which are used to reference components of values of tuple types (in a similar way as attributes in relational database schemas are used to reference fields of tuples).

Classical types are either atomic types or complex types constructed from atomic types and attributes by using the set and the tuple constructor. We formally define *classical types* by induction as follows:

- Every classical atomic type from $\mathcal{T}$ and every temporal atomic type is a classical type.
- If τ is a classical type, then $\{\tau\}$ is a classical type (called *classical set type*).
- If $A_1, \ldots, A_k$ are pairwise distinct attributes from $\mathcal{A}$ and $\tau_1, \ldots, \tau_k$ are classical types,
  then $[A_1 \colon \tau_1, \ldots, A_k \colon \tau_k]$ is a classical type (called *classical tuple type*).

We give some examples of classical types.

Example 9.2.1. Consider an application maintaining information about how long it takes for packages to get from one location to another. Such an application may be used by a package delivery service like DHL, Fedex, or UPS. The attributes Origin and Destination may be defined over the classical atomic type string, while the attribute Contents may be defined over the classical set type $\{\mathsf{string}\}$. A classical tuple type is $[\mathsf{Origin} : \mathsf{string}, \mathsf{Destination} : \mathsf{string}, \mathsf{Contents} : \{\mathsf{string}\}]$. □

The *values* of classical types are inductively defined as follows:

- For all classical atomic types $\tau \in \mathcal{T}$, every $v \in \mathrm{dom}(\tau)$ is a value of the classical type τ.
- If $v_1, \ldots, v_k$ are values of τ, then $\{v_1, \ldots, v_k\}$ is a value of the classical type $\{\tau\}$.
- If $A_1, \ldots, A_k$ are pairwise distinct attributes from $\mathcal{A}$ and $v_1, \ldots, v_k$ are values of $\tau_1, \ldots, \tau_k$, then $[A_1 \colon v_1, \ldots, A_k \colon v_k]$ is a value of the classical type $[A_1 \colon \tau_1, \ldots, A_k \colon \tau_k]$.

Some values of classical types in the Package Example are shown below.

Example 9.2.2. Boston is a value of the classical atomic type string, while $\{\mathsf{pens}, \mathsf{books}, \mathsf{camera}\}$ is a value of $\{\mathsf{string}\}$. Moreover, [Origin: Boston, Destination: New_York, Contents: $\mathsf{pens}, \mathsf{books}, \mathsf{camera}\}$] is a value of [Origin: string, Destination: string, Contents: $\{\mathsf{string}\}$]. □

Observe that the above classical types can be easily extended to also include the type constructors for lists (i.e., ordered sets) and bags (i.e., multisets) in addition to the set and tuple constructors.

9.3 Probabilistic Types

Probabilistic types are used to encode probabilistic information. They are either atomic probabilistic types, or complex probabilistic types constructed from classical types and atomic probabilistic types using the tuple constructor. We formally define *probabilistic types* by induction as follows (observe that the set of all probabilistic types includes all classical tuple types):

- If τ is a classical type, then $[\![\tau]\!]$ is a probabilistic type (called *atomic probabilistic type*).
- If $A_1, \ldots, A_k$ are pairwise distinct attributes from $\mathcal{A}$ and $\tau_1, \ldots, \tau_k$ are either classical or probabilistic types, then $[A_1 \colon \tau_1, \ldots, A_k \colon \tau_k]$ is a probabilistic type (called *probabilistic tuple type*). We call the attributes $A_1, \ldots, A_k$ its *top-level attributes*.

The following example illustrates the concept of a probabilistic type.

Example 9.3.1. In the Package Example, the attributes Delivery and STOPone may be defined over the atomic probabilistic type $[\![\mathsf{time}]\!]$ and the probabilistic tuple type $[\mathsf{City}\colon \mathsf{string},\ \mathsf{Arrive}\colon \mathsf{string},\ \mathsf{Shipment}\colon [\![\mathsf{time}]\!]]$, respectively, where time is a calendar. □

The values of probabilistic types are appropriately typed random variables. We define *values* of probabilistic types by induction as follows:

- A value of an atomic probabilistic type $[\![\tau]\!]$ is a finite set of pairs $(v, [l, u])$, where v is a value of τ, and l, u are reals with $0 \leq l \leq u \leq 1$.
- A value of a probabilistic type $[A_1\colon \tau_1, \ldots, A_k\colon \tau_k]$ is of the form $[A_1\colon v_1, \ldots, A_k\colon v_k]$, where $v_1, \ldots, v_k$ are values of $\tau_1, \ldots, \tau_k$. Given a value $v = [A_1\colon v_1, \ldots, A_k\colon v_k]$, we write $v.A_i$ to denote v_i.

Intuitively, a probabilistic value $v = \{(v_1, [l_1, u_1]), \ldots, (v_n, [l_n, u_n])\}$ says that v's value is exactly one member of from the set $\{v_1, \ldots, v_n\}$. The probability that v's value is v_i lies in the interval $[l_i, u_i]$. This is illustrated below.

Example 9.3.2. Let time be the calendar over the linear temporal hierarchy $hour \sqsupseteq minute$. An value of the atomic probabilistic type $[\![\mathsf{time}]\!]$ is $\{((12,30),[.4,.6]),((12,35),[.4,.6])\}$. Intuitively it says that this value is either $(12,30)$ or $(12,35)$. The probability that the value is $(12,30)$ is 0.4–0.6 and similarly for $(12,35)$. Similarly, an explicit value for the atomic probabilistic type $[\![\mathsf{string}]\!]$ is $\{(\mathsf{New\_York},[.3,.4]),(\mathsf{Washington},[.5,.6])\}$. □

Notice that for a probabilistic value $v = \{(v_1,[l_1,u_1]),\ldots,(v_n,[l_n,u_n])\}$ to be consistent, there must be some way of assigning point probabilities to the v_i's so that the above constraints are satisfied. For example, if $v = \{(5,[1,1]),(6,[1,1])\}$ then this is inconsistent (as it says that v equals both 5 and 6 with probability 1 which is impossible. Thus, we say that v is *consistent* iff there exists a probability function Pr ([148]) on $\{v_1,\ldots,v_n\}$ (that is, a mapping $Pr\colon \{v_1,\ldots,v_n\} \to [0,1]$ such that all $Pr(v_i)$ sum up to 1) such that $Pr(v_i) \in [l_i,u_i]$ for all $i \in \{1,\ldots,n\}$. It is not difficult to see that such a Pr exists iff $l_1 + \cdots + l_n \leq 1 \leq u_1 + \cdots + u_n$.

We can extend the above definition naturally to obtain the following notion of consistency for values of probabilistic types. A value $v = \{(v_1,[l_1,u_1]),\ldots,(v_n,[l_n,u_n])\}$ of an atomic probabilistic type is *consistent* iff $v_1,\ldots,v_n$ are pairwise distinct and $l_1 + \cdots + l_n \leq 1 \leq u_1 + \cdots + u_n$. A value v of a probabilistic type is *consistent* iff all values of atomic probabilistic types that occur in v are consistent.

Probabilistic Strategies We now define the concept of a probabilistic conjunction (resp., disjunction, difference) strategy, which is used in our algebraic operations to compute the probability interval of a conjunction (resp., disjunction, difference) of two pieces of information, which are each associated with a given probability interval.

Consider two events e_1 and e_2, which have a probability in the intervals $[l_1,u_1]$ and $[l_2,u_2]$, respectively. To compute the probability interval associated with the compound events $e_1 \wedge e_2$, $e_1 \vee e_2$, and $e_1 \wedge \neg e_2$, we need to know the dependencies between e_1 and e_2 (or lack thereof). For instance, e_1 and e_2 may be mutually exclusive, or probabilistically independent, or positively correlated (when e_1 implies e_2, or e_2 implies e_1), or we may be ignorant of the relationship between e_1 and e_2. Each of these situations yields a different way of computing the probability of $e_1 \wedge e_2$, $e_1 \vee e_2$, and $e_1 \wedge \neg e_2$. More formally, let $\mathcal{U}$ be the set of all nonempty subintervals $[l,u]$ of the unit interval $[0,1]$. Assume that the probabilities of the events e_1 and e_2 are in the intervals

$[l_1, u_1]$ **and** $[l_2, u_2]$**, respectively. A *conjunction* (resp., *disjunction*, *difference*) *strategy* is a function** $\otimes : \mathcal{U}^2 \to \mathcal{U}$ **(resp.,** $\oplus : \mathcal{U}^2 \to \mathcal{U}$**,** $\ominus : \mathcal{U}^2 \to \mathcal{U}$**) that computes the probability interval of** $e_1 \wedge e_2$ **(resp.,** $e_1 \vee e_2$**,** $e_1 \wedge \neg e_2$**) for some fixed dependencies between** e_1 **and** e_2 **(or lack thereof).**

Lakshmanan et al. [117] give axioms that conjunction and disjunction strategies should satisfy, but we do not repeat these here, except to say that our conjunction and disjunction strategies should also satisfy such axioms. Tables 9.1 and 9.2 show some examples of conjunction, disjunction, and difference strategies (see [69] for more examples). For associative and commutative conjunction (resp., disjunction) strategies $\odot$ **and nonempty intervals** $[l_1, u_1], \ldots, [l_k, u_k] \subseteq [0, 1]$ **with** $k \geq 1$**, we use** $\bigodot_{i=1}^{k} [l_i, u_i]$ **to denote** $[l_1, u_1] \odot \cdots \odot [l_k, u_k]$**. For** $k = 0$**, we define** $\bigodot_{i=1}^{k} [l_i, u_i]$ **as the constants** $[1, 1]$ **(resp.,** $[0, 0]$**).**

Table 9.1: Conjunction strategies

Mutual exclusion	$([l_1, u_1] \otimes_{me} [l_2, u_2]) = [0, 0]$
Positive correlation	$([l_1, u_1] \otimes_{pc} [l_2, u_2]) = [\min(l_1, l_2), \min(u_1, u_2)]$
Independence	$([l_1, u_1] \otimes_{in} [l_2, u_2]) = [l_1 \cdot l_2,\ u_1 \cdot u_2]$
Ignorance	$([l_1, u_1] \otimes_{ig} [l_2, u_2]) = [\max(l_1, l_2), \min(1, u_1 + u_2)]$

Table 9.2: Disjunction strategies

Mutual exclusion	$([l_1, u_1] \oplus_{me} [l_2, u_2]) = [\min(1, l_1 + l_2), \min(1, u_1 + u_2)]$
Positive correlation	$([l_1, u_1] \oplus_{pc} [l_2, u_2]) = [\max(l_1, l_2), \max(u_1, u_2)]$
Independence	$([l_1, u_1] \oplus_{in} [l_2, u_2]) = [l_1 + l_2 - l_1 \cdot l_2,\ u_1 + u_2 - u_1 \cdot u_2]$
Ignorance	$([l_1, u_1] \oplus_{ig} [l_2, u_2]) = [\max(0, l_1 + l_2 - 1), \min(u_1, u_2)]$

Table 9.3: Difference strategies

Mutual exclusion	$([l_1, u_1] \ominus_{me} [l_2, u_2]) = [l_1, \min(u_1, 1 - l_2)]$
Positive correlation	$([l_1, u_1] \ominus_{pc} [l_2, u_2]) = [\max(0, l_1 - u_2), \max(0, u_1 - l_2)]$
Independence	$([l_1, u_1] \ominus_{in} [l_2, u_2]) = [l_1 \cdot (1 - u_2),\ u_1 \cdot (1 - l_2)]$
Ignorance	$([l_1, u_1] \ominus_{ig} [l_2, u_2]) = [\max(0, l_1 - u_2), \min(u_1, 1 - l_2)]$

Chapter 10

Temporal Probabilistic Object Bases

In this Chapter, we first introduce the concept of a schema for temporal probabilistic object bases. Intuitively, a schema consists of two parts. The first is a hierarchy of classes with associated types. The second part specifies a conditional probability. If c_1 is an immediate subclass of c_2, this conditional probability specifies the probability of a member of c_2 being a member of c_1. For example, c_1 could be the class "registered_letters" and c_2 could be the class "letters." In this case, we may have a probability of 0.05 that an arbitrary letter is a registered letter.

Second, we define the inheritance completion of a schema, which adds to every class, all the attributes (with their types) that are inherited from superclasses. Finally, we introduce temporal probabilistic object base instances with respect to this inheritance completion schema.

10.1 Temporal Probabilistic Object Base Schema

We now define the concept of a temporal probabilistic object base schema. Informally, every schema specifies a finite set of classes $\mathcal{C}$. Every class $c \in \mathcal{C}$ has an associated probabilistic tuple type $\sigma(c)$ specifying the type of objects in this class. Moreover, every class $c \in \mathcal{C}$ is associated with a partition $\mathrm{me}(c)$ of the set of all its immediate subclasses into sets (or clusters) of pairwise disjoint classes. For example, $\mathrm{me}(c) = \{\{c_1, c_2\}, \{c_3, c_4, c_5\}\}$ says that $c_1, \ldots, c_5$ are the immediate subclasses of c, and that an object o that belongs to the class c can belong to at most one class among c_1 and c_2, and to at most one class among c_3, c_4, and c_5. Finally, every immediate subclass relationship

between two classes c_1 and c_2 is associated with the conditional probability that an arbitrary object belongs to c_1 given that it belongs to c_2.

Formally, a *temporal probabilistic object base schema* (or *TPOB-schema*) $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ consists of (i) a finite set of *classes* $\mathcal{C}$, (ii) a *type assignment* σ that associates with each class $c \in \mathcal{C}$ a probabilistic tuple type, (iii) a mapping me that associates with each class $c \in \mathcal{C}$ a partition of the set of all its *immediate subclasses* $d \in \mathcal{C}$, and (iv) a *probability assignment* $\wp$ that associates with every pair of classes $(c_i, c) \in \mathcal{C} \times \mathcal{C}$, where c_i is an immediate subclass of c, a positive rational number in $[0, 1]$ such that $\Sigma_{c_i \in \mathcal{P}} \wp(c_i, c) \leq 1$ for every class $c \in \mathcal{C}$ and every cluster $\mathcal{P} \in \mathrm{me}(c)$. Observe that me defines a directed graph $(\mathcal{C}, \Rightarrow)$, where $c_1 \Rightarrow c_2$ iff c_1 is an immediate subclass of c_2. As usual, we assume that $(\mathcal{C}, \Rightarrow)$ is acyclic. The following example illustrates the concept of a TPOB-schema.

Example 10.1.1. The TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ for the Package Example is given as follows:

- $\mathcal{C} = \{\mathsf{Package}, \mathsf{Letter}, \mathsf{Box}, \mathsf{Tube}, \mathsf{Priority}, \mathsf{Express\_saves}, \mathsf{One\text{-}transfer}, \mathsf{Two\text{-}transfer}\}$;
- the type assignment σ is given by Table 10.1 below;
- $\mathrm{me}(\mathsf{Package}) = \{\{\mathsf{Letter}, \mathsf{Box}, \mathsf{Tube}\}, \{\mathsf{Priority}, \mathsf{Express\_saves}\}\}$,
  $\mathrm{me}(\mathsf{Box}) = \mathrm{me}(\mathsf{Express\_saves}) = \{\{\mathsf{Two\text{-}transfer}\}\}$,
  $\mathrm{me}(\mathsf{Tube}) = \mathrm{me}(\mathsf{Priority}) = \{\{\mathsf{One\text{-}transfer}\}\}$,
  $\mathrm{me}(\mathsf{Letter}) = \mathrm{me}(\mathsf{One\text{-}transfer}) = \mathrm{me}(\mathsf{Two\text{-}transfer}) = \emptyset$;
- the probability assignment $\wp$ is given in Fig. 10.1.

Note that the acyclic directed graph $(\mathcal{C}, \Rightarrow)$ is the graph resulting from Fig. 10.1 when the d-nodes are contracted to Package and the probability labels are removed. The d-nodes of Fig. 10.1 denote disjoint decompositions – every package is either a letter, box, or tube, and either an express package or a priority package. □

We now introduce some concepts related to TPOB-schemas. A *top-level attribute* of a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ is a top-level attribute (as defined in Section 9.3) of some $\sigma(c)$ where $c \in \mathcal{C}$. A *directed path* in $(\mathcal{C}, \Rightarrow)$ is a sequence of classes $c_1, \ldots, c_k \in \mathcal{C}$ such that $c_1 \Rightarrow \cdots \Rightarrow c_k$ and $k \geq 1$. We denote by $\Rightarrow^\star$ the reflexive and transitive closure of $\Rightarrow$. We say c_1 is a *subclass* (resp., *strict subclass*) of c_2, or c_2 is a *superclass* (resp., *strict superclass*) of c_1, iff $c_1 \Rightarrow^\star c_2$ (resp., $c_1 \Rightarrow^\star c_2$ and $c_1 \neq c_2$). A class $d \in \mathcal{C}$

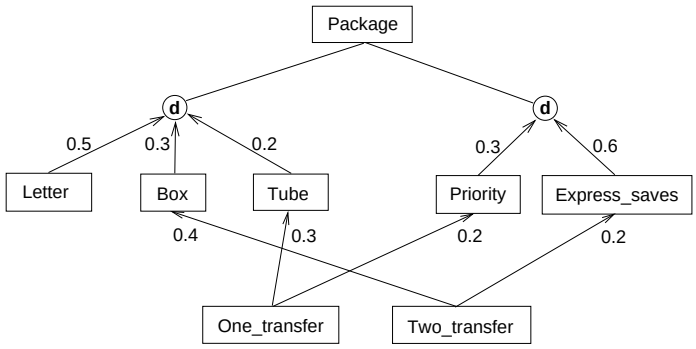


Figure 10.1: Package Example with probability assignment $\wp$

Table 10.1: Type assignment σ

c	$\sigma(c)$
Package	[Origin: string, Destination: string, Delivery: [[time]]]
Letter	[Height: float, Width: float]
Box	[Height: float, Width: float, Depth: float, Contents: {string}]
Tube	
Priority	[Time: [[time]]]
Express_saves	
One-transfer	[City: string, Arrive: [[time]], Shipment: [[time]]]
Two-transfer	[STOPone: [City: string, Arrive: [[time]], Shipment: [[time]]], STOPtwo: [City: string, Arrive: [[time]], Shipment: [[time]]]]

is *minimal* under $\Rightarrow^\star$ in a set of classes $\mathcal{D} \subseteq \mathcal{C}$ iff $d \in \mathcal{D}$ and no class in $\mathcal{D}$ is a strict subclass of d.

We finally define the important notion of consistency of TPOB-schemas. Intuitively, a TPOB-schema is consistent iff each class c can be associated with a nonempty set $\zeta(c)$ of objects such that the immediate subclass and disjointness relationships and the relative cardinalities expressed by me and $\wp$, respectively, are satisfied. If there is no way to make this happen, then this means that there is some fundamental problem with the conditional probability assignments to the TPOB-schema. Every TPOB-schema should have this property. Formally, an *interpretation* $\mathcal{I} = (\mathcal{O}, \zeta)$ of a TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ consists of a nonempty set $\mathcal{O}$, and a mapping ζ from $\mathcal{C}$ to the set of all finite subsets of $\mathcal{O}$. The interpretation $\mathcal{I}$ is a *taxonomic model* of S iff the following conditions are satisfied:

C1 $\zeta(c) \neq \emptyset$**, for all classes** $c \in \mathcal{C}$**.**

C2 $\zeta(c) \subseteq \zeta(d)$**, for all classes** $c, d \in \mathcal{C}$ **with** $c \Rightarrow d$**.**

C3 $\zeta(c) \cap \zeta(d) = \emptyset$**, for all distinct classes** $c, d \in \mathcal{C}$ **in the same cluster** $\mathcal{P} \in \bigcup \mathrm{me}(\mathcal{C})$**.**

The first two axioms say that classes must not be empty and that the objects in a subclass must be a subset of the objects in a superclass. To see why the third axiom is present, consider the fact (from Figure 10.1) that a package is either a tube, box, or letter — this axiom forces a package to be exactly one of these three (e.g. a package cannot be both a tube and a box).

The interpretation $\mathcal{I}$ **is a *taxonomic and probabilistic model* (or *model*) of S iff it is a taxonomic model of S and for all classes** $c, d \in \mathcal{C}$ **with** $c \Rightarrow d$**, axiom (C4) below holds:**

C4 $|\zeta(c)| = \wp(c, d) \cdot |\zeta(d)|$**.**

We say S is *consistent* iff a model of S exists. We say that two classes $c, d \in \mathcal{C}$ **are *taxonomically disjoint* (or *t-disjoint*) iff** $\zeta(c) \cap \zeta(d) = \emptyset$ **for every taxonomic model** $\mathcal{I} = (\mathcal{O}, \zeta)$ **of S.**

The axiom above says that the number of items in each class must be consistent with the conditional probability labeling. For example, if an interpretation assigns 100 **objects to the class "Package", then it must assign 50 objects to the class "Letter" — if not, it will not satisfy the probability requirement that half the packages are letters.**

Note that the work in this section builds upon definitions in [68]. There, it is shown that deciding the consistency of probabilistic object base schemas is NP-complete — this result also applies here. However, there are important special cases of TPOB-schemas, which can be tested in polynomial time, and for which deciding consistency can also be done in polynomial time (see [68] for detailed algorithms).

10.2 Inheritance Completion

The inheritance completion of a TPOB-schema adds to the type of every class $c \in \mathcal{C}$**, all the attributes (with their types) that are inherited from all superclasses of** c**. We use inheritance strategies to specify how to resolve conflicts that arise due to multiple inheritance. More precisely, we add to the type of each class** $c \in \mathcal{C}$**, every top-level attribute** A **with its type at a minimal superclass of** c**. If more than one such minimal superclass exists, then we have a conflict due to multiple inheritance, and we**

use an inheritance strategy to select a unique class among the set of all such minimal superclasses.

More formally, let $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ be a TPOB-schema, and let $\boldsymbol{A}$ denote the set of all its top-level attributes. Let $\mathsf{min}_{\mathbf{S}} \colon \mathcal{C} \times \boldsymbol{A} \to \mathcal{C}$ be the mapping that assigns to each pair $(c, A) \in \mathcal{C} \times \boldsymbol{A}$ the set of all minimal classes under $\Rightarrow^{\star}$ in the set of all superclasses $d \in \mathcal{C}$ of c such that A is a top-level attribute of $\sigma(d)$. An *inheritance strategy* for $\mathbf{S}$ is a partial mapping $\mathsf{inh}_{\mathbf{S}} \colon \mathcal{C} \times \boldsymbol{A} \to \mathcal{C}$ that assigns a class $d \in \mathsf{min}_{\mathbf{S}}(c, A)$ to every pair $(c, A) \in \mathcal{C} \times \boldsymbol{A}$ such that $\mathsf{min}_{\mathbf{S}}(c, A) \neq \emptyset$.

The inheritance completion of a TPOB-schema is obtained by adding to each type of a class, all top-level attributes with their types that are inherited from minimal superclasses. More formally, the *inheritance completion TPOB-schema* of a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ is the TPOB-schema $\mathbf{S}^{\star} = (\mathcal{C}, \sigma^{\star}, \mathrm{me}, \wp)$, where $\sigma^{\star}(c) = [A_1 \colon \tau_1, \ldots, A_k \colon \tau_k]$ such that (i) $A_1, \ldots, A_k$ are all $A \in \mathcal{A}$ such that $\mathsf{inh}_{\mathbf{S}}(c, A)$ is defined, and (ii) $\sigma(\mathsf{inh}_{\mathbf{S}}(c, A_i))$ has the type τ_i at A_i, for all $i \in \{1, \ldots, k\}$. If $\mathbf{S} = \mathbf{S}^{\star}$, then $\mathbf{S}$ is *fully inherited*. The following example shows the inheritance completion of the TPOB-schema in our Package Example.

Example 10.2.1. Consider the TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ of Example 10.1.1. Its inheritance completion TPOB schema is $\mathbf{S}^{\star} = (\mathcal{C}, \sigma^{\star}, \mathrm{me}, \wp)$, where $\sigma^{\star}$ is given in Table 10.2. □

Throughout the rest of this paper, we assume that all TPOB-schemas are fully inherited.

10.3 Temporal Probabilistic Object Base Instance

This section introduces the notion of a temporal probabilistic object base (TPOB) instance. Throughout this thesis, we assume a countably infinite set $\mathcal{O}$ of *object identifiers* (or *oids*) (we will often abuse notation and call *oids* objects). Note that we assume that $\mathcal{O}$ is countably infinite to ensure that we may have arbitrary large finite TPOB-instances. For the algebraic operations of natural join, Cartesian product, and conditional join (see Section 12), we additionally assume that $\mathcal{O}$ is closed under Cartesian product, that is, $\mathcal{O} \times \mathcal{O} \subseteq \mathcal{O}$.

A *temporal probabilistic object base instance* (or *TPOB-instance*) $\mathbf{I} = (\pi, \nu)$ over a consistent TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ consists of (i) a mapping π that assigns to

Table 10.2: Type assignment $\sigma^\star$ of the inheritance completion TPOB-schema

c	$\sigma^\star(c)$
Package	[Origin: string, Destination: string, Delivery: [[time]]]
Letter	[Origin: string, Destination: string, Delivery: [[time]], Height: float, Width: float]
Box	[Origin: string, Destination: string, Delivery: [[time]], Height: float, Width: float,
Tube	Depth: float, Contents: {string}]
Priority	[Origin: string, Destination: string, Delivery: [[time]], Time: [[time]]]
Express_saves	
One-transfer	[Origin: string, Destination: string, Delivery: [[time]], Height : float, Width : float, Depth : float, Contents : {string}, Time : [[time]], City: string, Arrive: [[time]], Shipment : [[time]]]
Two-transfer	[Origin: string, Destination: string, Delivery: [[time]], Height: float, Width: float, Depth: float, Contents: {string}, Time: [[time]], STOPone: [City: string, Arrive: [[time]], Shipment: [[time]]], STOPtwo: [City: string, Arrive: [[time]], Shipment: [[time]]]]

each class $c \in \mathcal{C}$ a finite subset of $\mathcal{O}$ such that $\pi(c_1) \cap \pi(c_2) = \emptyset$ for all distinct classes $c_1, c_2 \in \mathcal{C}$, and (ii) a mapping ν that assigns to each oid $o \in \pi(c)$, $c \in \mathcal{C}$, a value of type $\sigma^\star(c)$. We write $\pi(\mathcal{C})$ to refer to the set $\bigcup\{\pi(c) \mid c \in \mathcal{C}\}$ of all oids in I. For every $c \in \mathcal{C}$, we use $\pi^\star(c)$ to denote the set $\bigcup\{\pi(c') \mid c' \in \mathcal{C}, c' \Rightarrow^\star c\}$ of all oids that *belong* to c, which is to be distinguished from the set $\pi(c)$ of all oids that are *created* in c. Intuitively, $\pi(c)$ is the set of oids in class c, while $\nu(o)$ specifies the value of an object $o \in \pi(c)$. The following example shows a TPOB-instance over the TPOB-schema in our Package Example.

Example 10.3.1. Tables 10.3 and 10.4 show a TPOB-instance $\mathbf{I}=(\pi, \nu)$ over the TPOB-schema **S** of Example 10.1.1. According to this, o_3 is a priority package, o_5 is a two-transfer package. o_3 is being shipped from Rome to Boston and its expected delivery time is either $(18, 00)$ or $(18, 31)$. The probability of the former is 40–60% while that of the latter is 30–50%. □

We now define the concept of a probabilistic extent of a class. In classical object bases, the extent of a class c is a mapping $\mathrm{ext(c)}$ that associates with every object o in the object base the number 0 (resp., 1) iff o does not belong (resp., does belong) to the class c. In probabilistic object bases, the probabilistic extent of a class c is a mapping

Table 10.3: The mappings π and π^*

c	$\pi(c)$	$\pi^*(c)$
Package	$\{\}$	$\{o_3, o_5\}$
Letter	$\{\}$	$\{\}$
Box	$\{\}$	$\{o_5\}$
Tube	$\{\}$	$\{\}$

c	$\pi(c)$	$\pi^*(c)$
Priority	$\{o_3\}$	$\{o_3\}$
Express_saves	$\{\}$	$\{o_5\}$
One-transfer	$\{\}$	$\{\}$
Two-transfer	$\{o_5\}$	$\{o_5\}$

Table 10.4: Value assignment ν

o	$\nu(o)$
o_3	[Origin: Rome, Destination: Boston, Delivery: $\{((18,00),[.4,.6]),((18,31),[.3,.5])\}$, Time: $\{((8,00),[.45,.5]),((8,10),[.4,.5])\}$]
o_5	[Origin: Paris, Destination: San_Jose, Delivery: $\{((12,00),[.5,.7]),((12,15),[.4,.5])\}$, Height: 60, Width: 50, Depth: 40, Contents: {photos, books}, Time: $\{((12,00),[.3,.4]),((12,05),[.4,.7])\}$, STOPone: [City: New_York, Arrive: $\{((14,00),[.3,.5]),((14,30),[.7,.8])\}$, Shipment: $\{((16,00),[1,1])\}$], STOPtwo: [City: ST_Louis, Arrive: $\{((17,30),[.2,.6]),((17,45),[.5,.6])\}$, Shipment: $\{((18,00),[.3,.5]),((18,30),[.6,.7])\}$]]

$\mathrm{ext(c)}$ that associates with every object o in the object base the possible probabilities with which o belongs to c. If o belongs to a subclass of c (resp., a class c' t-disjoint to c), then obviously $\mathrm{ext(c)}(o)=\{1\}$ (resp., $\mathrm{ext(c)}(o)=\{0\}$). Otherwise, $\mathrm{ext(c)}(o)$ is the set of all products of probabilities from c up to a minimal superclass c' of c that contains o. More formally, let $\mathbf{I}=(\pi,\nu)$ be a TPOB-instance over a consistent TPOB-schema $\mathbf{S}=(\mathcal{C},\sigma,\mathrm{me},\wp)$, and let c be a class from $\mathcal{C}$. The *probabilistic extent* of c, denoted $\mathrm{ext(c)}$, maps each oid $o\in\pi(\mathcal{C})$ to a *set* of rational numbers in $[0,1]$ as follows:

(1) If $o\in\pi^\star(c)$, then $\mathrm{ext(c)}(o)=\{1\}$.

(2) If $o\in\pi^\star(c')$ with a class $c'\in\mathcal{C}$ that is t-disjoint from c, then $\mathrm{ext(c)}(o)=\{0\}$.

(3) Otherwise, $\mathrm{ext(c)}(o)=\{p\mid p$ is the product of the edge probabilities on a path from c up to a minimal class c' under $\Rightarrow^\star$ in the set of all superclasses $d\in\mathcal{C}$ of c such that $o\in\pi^\star(d)\}$.

We call the class c in (1), the class c' in (2), and every class c' as in (3) a *characteristic class for* $\mathrm{ext(c)}(o)$. The following example illustrates the concept of a probabilistic extent of a class.

Example 10.3.2. The probabilistic extent of the class One-transfer in our Package Example maps the oids o_3 and o_5 to the sets of rational numbers $\{.2\}$ and $\{0\}$, respectively. □

If the probabilistic extent of every class c assigns a unique probability to every object o, then we say that the underlying TPOB-instance is coherent. This is important: intuitively, given an object o and a class c, there must be only one probability that o belongs to c (otherwise something would be wrong). For example, given that o is a package in Figure 10.1, we know that the probability of its having one transfer is 0.06 — notice that Figure 10.1 has two paths and the computed probability across *both* paths is 0.06. This is the kind of intuition that the notion of *coherence* below attempts to capture. Formally, a TPOB-instance $\mathrm{I} = (\pi, \nu)$ over a consistent TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ is *coherent* iff for every class $c \in \mathcal{C}$ and every object $o \in \pi(\mathcal{C})$, the probabilistic extent $\mathrm{ext(c)}(o)$ contains *at most* one element.

Another important concept for TPOB-instances is that of consistency, which is simply an extension of the concept of consistency for values of probabilistic types. A TPOB-instance is consistent iff all its values of probabilistic types are consistent. Inconsistent values of probabilistic types and inconsistent TPOB-instances are meaningless. Hence, the notion of consistency is very important. Formally, a TPOB-instance $\mathrm{I} = (\pi, \nu)$ over a consistent TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ is *consistent* iff $\nu(o)$ is consistent for all $o \in \pi(\mathcal{C})$.

Chapter 11

TPOB-Algebra: Unary Operations

In this Chapter, we introduce the unary operations of the TPOB-algebra. These operations take a TPOB-instance over a TPOB-schema as input, and produce a TPOB-instance over a TPOB-schema as output. We describe the unary operations of selection, restricted selection, renaming, projection, and extraction. *Unless specified otherwise, we assume that all input TPOB-schemas are fully inherited*.

11.1 Selection

The selection condition finds all objects in a TPOB-instance I which satisfy a probabilistic selection condition ϕ. We will formally define a probabilistic selection condition in this section. A simple probabilistic selection condition says that ordinary (i.e., non-probabilistic) selection conditions must be true with probability inside a given range. Boolean combinations of such simple probabilistic selection conditions are also allowed. A selection condition is a logical combination of atomic selection conditions. An atomic selection condition may include membership of an object in a class membership, or a comparison that involves attribute values of an object. Path expressions (defined below) refer to "inner" attribute values of an object.

Hence, we first define path expressions and atomic selection conditions, which are then used to construct selection conditions and probabilistic selection conditions. In the rest of this section, let $\mathrm{I} = (\pi, \nu)$ be a TPOB-instance over a TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$.

11.1.1 Path Expressions

Path expressions are finite sequences of attributes, which refer to "inner" attribute values of an object. We formally define *path expressions* by induction as follows. A path expression for the tuple type $[A_1 \colon \tau_1, \ldots, A_k \colon \tau_k]$ has the form A_i or the form $A_i.P_i$, where P_i is a path expression for τ_i. A path expression for the atomic probabilistic type $[\![\tau]\!]$ is of the form $[\![P]\!]$, where P is a path expression for τ. The following shows some examples of path expressions.

Example 11.1.1. STOPone, STOPone.City, and STOPone.Arrive.[[time]] are path expressions for the probabilistic tuple type [STOPone: [City: string, Arrive: [[time]], Shipment: [[time]]]]. □

We now define how a path expression P addresses a part of a value v. Given a path expression P for type τ, the *valuation* of P under a value v of τ, denoted $v.P$, is inductively defined by:

- **if** $v = [A_1 \colon v_1, \ldots, A_k \colon v_k]$ **and** $P = A_i$**, then** $v.P = v_i$**;**
- **if** $v = [A_1 \colon v_1, \ldots, A_k \colon v_k]$ **and** $P = A_i.R$**, then** $v.P = v_i.R$**;**
- **if** $v = \{(v_1, I_1), \ldots, (v_k, I_k)\}$ **and** $P = [\![R]\!]$**, then** $v.P = \{(v_1, I_1, R), \ldots, (v_k, I_k, R)\}$**. We call such sets** $\{(v_1, I_1, R), \ldots, (v_k, I_k, R)\}$ ***generalized values*** **of** τ**.**

Otherwise, $v.P$ is undefined. The following example illustrates this concept of valuation.

Example 11.1.2. The valuation of the path expression STOPone.City under the value [STOPone: [City: Boston, Arrive: $\{((8, 00), [1, 1])\}$, Shipment: $\{((9, 00), [1, 1])\}$]] is given by Boston. □

11.1.2 Atomic Selection Conditions

An atomic selection condition describes either (i) a class membership of an object, or (ii) a comparison between an attribute value of an object and a constant value, or (ii) a comparison between two attribute values of an object. More formally, an *atomic selection condition* has one of the following forms:

- $in(c)$**, where** c **is a class in** $\mathcal{C}$**;**
- $P\ \theta\ v$**, where** P **is a path expression,** v **is a value, and** $\theta \in \{\leq, <, =, \neq, >, \geq\}$**;**

- $P_1\,\theta_{\otimes}\,P_2$**, where** P_1, P_2 **are path expressions,** $\otimes$ **is a conjunction strategy, and** $\theta \in \{\leq, <, =, \neq, >, \geq\}$**.**

We give some examples of atomic selection conditions in the Package Example.

Example 11.1.3. $in(\mathsf{Letter})$ is an atomic selection condition which specifies the set of all objects that are letters. Likewise, $\mathsf{STOPone.City} = \mathsf{New\_York}$ is an atomic selection condition which specifies the set of "all objects that have New York as the first stop." $\mathsf{Origin} =_{\otimes_{in}} \mathsf{Destination}$ is an atomic selection condition which specifies all objects whose Origin and Destination are the same, assuming the value of the Origin is independent of the value of the Destination. □

We now define the meaning of an atomic selection condition ϕ **under an object** o **in I. We associate with each such** ϕ **a closed subinterval of** $[0, 1]$**, which describes the range for the probability that the object** o **in I satisfies** ϕ**. More formally, the *probabilistic valuation* of** ϕ **with respect to I and** $o \in \pi(\mathcal{C})$**, denoted** $\mathrm{prob}_{\mathbf{I},o}(\phi)$**, is defined as follows (where** $\oplus$ **is the disjunction strategy for mutual exclusion):**

- $\mathrm{prob}_{\mathbf{I},o}(in(c)) = [\min(\mathrm{ext(c)}(o)), \max(\mathrm{ext(c)}(o))]$**.**
- **Let** P **be a path expression for the type of** o**. If** $\nu(o).P$ **is a value of a classical type, then define** $V = \{(\nu(o), [1, 1], P)\}$**, else if** $\nu(o).P$ **is a generalized value of an atomic probabilistic type, then define** $V = \nu(o).P$**. Otherwise,** V **is undefined. Then,**

$$\mathrm{prob}_{\mathbf{I},o}(P\,\theta\,v) = \begin{cases} \bigoplus_{i=1}^{k} I_i & \textbf{if } V \textbf{ is defined} \\ \textbf{undefined} & \textbf{otherwise,} \end{cases}$$

  where $I_1, \ldots, I_k$ **are the intervals** I **such that** $(w, I, S) \in V$ **and** $w.S\,\theta\,v$**, if** V **is defined. Note that** $\mathrm{prob}_{\mathbf{I},o}(P\,\theta\,v)$ **is undefined if some** $w.S\,\theta\,v$ **is undefined.**

- **For each** $i \in \{1, 2\}$**, let** P_i **be a path expression for the type of** o**. If** $\nu(o).P_i$ **is a value of a classical type, then define** $V_i = \{(\nu(o), [1, 1], P_i)\}$**, else if** $\nu(o).P_i$ **is a generalized value of an atomic probabilistic type, then define** $V = \nu(o).P_i$**. Otherwise,** V_i **is undefined. Then,**

$$\mathrm{prob}_{\mathbf{I},o}(P_1\,\theta_{\otimes}\,P_2) = \begin{cases} \bigoplus_{i=1}^{k} I_i & \textbf{if } V_1 \textbf{ and } V_2 \textbf{ are defined} \\ \textbf{undefined} & \textbf{otherwise,} \end{cases}$$

where $I_1, \ldots, I_k$ are the intervals $I \otimes J$ such that $(v_1, I, S_1) \in V_1$, $(v_2, J, S_2) \in V_2$, and $v_1.S_1 \, \theta \, v_2.S_2$, if V_1 and V_2 are defined. Note that $\mathrm{prob}_{\mathbf{I},o}(P_1 \, \theta_{\otimes} \, P_2)$ is undefined, if some $v_1.S_1 \, \theta \, v_2.S_2$ is undefined.

The following example shows some probabilistic valuations of atomic selection conditions.

Example 11.1.4. $in(\mathsf{One\text{-}transfer})$ is assigned $[.2, .2]$ (resp., $[0, 0]$) under $\mathrm{prob}_{\mathbf{I},o_3}$ (resp., $\mathrm{prob}_{\mathbf{I},o_5}$), while $\mathsf{STOPone.City{=}New\_York}$ is undefined under $\mathrm{prob}_{\mathbf{I},o_3}$, and assigned $[1, 1]$ under $\mathrm{prob}_{\mathbf{I},o_5}$. □

11.1.3 Selection Conditions

Selection conditions **are logical combinations of atomic selection conditions. A formal inductive definition is as follows. (i) Every atomic selection condition is a selection condition. (ii) If ϕ and ψ are selection conditions and $\otimes$ (resp., $\oplus$) is a conjunction (resp., disjunction) strategy, then $\phi \wedge_{\otimes} \psi$ (resp., $\phi \vee_{\oplus} \psi$) is a selection condition. The following shows an example of a selection condition from the Package Example.**

Example 11.1.5. $in(\mathsf{One\text{-}transfer}) \wedge_{\otimes_{in}} (\mathsf{Arrive} < (13, 00) \vee_{\oplus_{in}} \mathsf{Arrive} > (14, 00))$ is a selection condition that specifies "all objects in One-transfer that arrive before 13:00 or after 14:00". □

We now define the meaning of a selection condition ϕ under an object o in I, by associating with each "well-formed" such ϕ a closed subinterval of $[0, 1]$, which describes the range for the probability that the object o in I satisfies ϕ. We do this by extending the probabilistic valuation $\mathrm{prob}_{\mathbf{I},o}$ as follows:

- $\mathrm{prob}_{\mathbf{I},o}(\phi \wedge_{\otimes} \psi) = \mathrm{prob}_{\mathbf{I},o}(\phi) \otimes \mathrm{prob}_{\mathbf{I},o}(\psi)$**;**
- $\mathrm{prob}_{\mathbf{I},o}(\phi \vee_{\oplus} \psi) = \mathrm{prob}_{\mathbf{I},o}(\phi) \oplus \mathrm{prob}_{\mathbf{I},o}(\psi)$**.**

The following example shows some probabilistic valuations of selection conditions.

Example 11.1.6. The selection condition $in(\mathsf{Priority}) \wedge_{\otimes_{in}} \mathsf{Time} < (14, 00)$ is assigned the probability intervals $[.95, 1]$ and $[0, 0]$ under $\mathrm{prob}_{\mathbf{I},o_3}$ and $\mathrm{prob}_{\mathbf{I},o_5}$, respectively. □

Probabilistic Selection Conditions

We define *probabilistic selection conditions* inductively as follows. If ϕ is a selection condition, and l, u are reals with $0 \leq l \leq u \leq 1$, then $(\phi)[l, u]$ is a probabilistic selection condition (an *atomic* one). If ϕ and ψ are probabilistic selection conditions, then so are $\neg\phi$ and $(\phi \wedge \psi)$. We use $(\phi \vee \psi)$ to abbreviate $\neg(\neg\phi \wedge \neg\psi)$. Some examples of probabilistic selection conditions for the Package Example are shown below.

Intuitively, a probabilistic selection condition of the form $(\phi)[l, u]$ says find all objects that satisfy condition ϕ with probability between l and u (both inclusive).

Example 11.1.7. $(\mathsf{Time} < (14, 00))[.5, 1]$ is a probabilistic selection condition, which specifies "all objects whose value in the attribute Time is smaller than 14:00 with a probability in $[.5, 1]$". □

We now define what it means for an object o in I to satisfy a probabilistic selection condition ϕ. The *satisfaction* of ϕ under I and o, denoted $\mathrm{prob}_{\mathbf{I},o} \models \phi$, is inductively defined by:

- $\mathrm{prob}_{\mathbf{I},o} \models (\phi)[l, u]$ **iff** $\mathrm{prob}_{\mathbf{I},o}(\phi) \subseteq [l, u]$**;**
- $\mathrm{prob}_{\mathbf{I},o} \models \neg\phi$ **iff it is not the case that** $\mathrm{prob}_{\mathbf{I},o} \models \phi$**;**
- $\mathrm{prob}_{\mathbf{I},o} \models \phi \wedge \psi$ **iff** $\mathrm{prob}_{\mathbf{I},o} \models \phi$ **and** $\mathrm{prob}_{\mathbf{I},o} \models \psi$**.**

The next example illustrates the satisfaction of probabilistic selection conditions.

Example 11.1.8. The following satisfaction (and non-satisfaction) relations hold in the Package Example:

- $\mathrm{prob}_{\mathbf{I},o_3} \not\models (in(\mathsf{Priority}) \wedge_{\otimes_{in}} \mathsf{Time} < (14, 00))[.96, 1]$,
- $\mathrm{prob}_{\mathbf{I},o_3} \models (in(\mathsf{Priority}) \wedge_{\otimes_{in}} \mathsf{Time} < (14, 00))[.5, 1]$,
- $\mathrm{prob}_{\mathbf{I},o_3} \models (in(\mathsf{Priority}) \wedge_{\otimes_{in}} \mathsf{Time} < (14, 00))[.5, 1] \wedge (in(\mathsf{One\text{-}transfer}))[.2, .3]$. □

11.1.4 Selection Operation

The selection operator finds all objects in a probabilistic instance I that satisfy some probabilistic selection condition. Formally, the *selection* on I with respect to a probabilistic selection condition ϕ, denoted $\sigma_\phi(\mathrm{I})$, is the TPOB-instance (π', ν') over S, where:

- $\pi'(c) = \{o \in \pi(c) \mid \models$ **is defined for** $\mathrm{prob}_{\mathbf{I},o}$ **and** ϕ**, and** $\mathrm{prob}_{\mathbf{I},o} \models \phi\}$**, for all** $c \in \mathcal{C}$**.**
- $\nu' = \nu \,|\, \pi'(\mathcal{C})$**.**

The following example illustrates the use of the selection operator in the Package Example.

Example 11.1.9. Consider the TPOB-instance $\mathbf{I} = (\pi, \nu)$ of Example 10.3.1 and the probabilistic selection condition $\phi = \neg(in(\mathsf{Priority}) \wedge_{\otimes_{in}} \mathsf{Time} < (14,00))[0,0]$. Then, $\sigma_\phi(\mathbf{I}) = (\pi', \nu')$ contains the set of all objects in $\mathbf{I}$ that belong to Priority and have a value in Time smaller than 14:00 with a positive probability. Observe that $\pi'(\mathsf{Priority}) = \{o_3\}$, $\pi'(c) = \emptyset$ for all other classes c, and ν' is given by Table 11.1. □

Table 11.1: ν' resulting from selection

o	$\nu'(o)$
o_3	[Origin: Rome, Destination: Boston, Delivery: $\{((18,00),[.4,.6]),((18,31),[.3,.5])\}$, Time: $\{((8,00),[.45,.5]),((8,10),[.4,.5])\}$]

11.2 Restricted Selection

Restricted selection is a new operator not present in the usual relational (or object) algebra. Consider an atomic selection condition ϕ **of the form** $P\,\theta\,w$**. Informally, this operation is a selection with respect to** ϕ **on the value** $v = \{(v_1, I_1), \ldots,$ $(v_k, I_k)\}$ **of an atomic probabilistic type "inside" the value** $\nu(o)$ **of every object** o**. As usual, the path expression** P **in** ϕ **is used to specify the part** v **of the value** $\nu(o)$**.**

We first formally define the restricted selection operation on values of probabilistic tuple types as follows. Let τ **be a probabilistic tuple type, and let** $v = [A_1 : v_1, \ldots, A_i : v_i, \ldots, A_k : v_k]$ **be a value of** τ**. Let** ϕ **be an atomic selection condition of the form** $P\,\theta\,w$**, where** P **is a path expression for** τ**. Then, the *restricted selection* on** v **w.r.t.** ϕ**, denoted** $\sigma^r_\phi(v)$**, is defined as follows:**

- **If** v_i **is a value of a classical type,** $P = A_i$**, and** $v_i\,\theta\,w$**, then** $\sigma^r_\phi(v) = v$**.**
- **If** v_i **is a value of an atomic probabilistic type, and** $P = A_i$**, then** $\sigma^r_\phi(v)$ **is obtained from** v **by replacing** v_i **by** $\{(w', I) \in v_i \mid w'\,\theta\,w\}$**.**

- **If v_i is a value of a probabilistic tuple type, and $P = A_i.R$, then $\sigma^r_\phi(v)$ is obtained from v by replacing v_i by $\sigma^r_{R\,\theta\,w}(v_i)$.**

Otherwise, $\sigma^r_\phi(v)$ is undefined.

We are now ready to extend the restricted selection operator to TPOB-instances as follows. Let $\mathbf{I}=(\pi,\nu)$ be a TPOB-instance over a TPOB schema $\mathbf{S}=(\mathcal{C},\sigma,\mathrm{me},\wp)$. Let ϕ be an atomic selection condition of the form $P\,\theta\,w$, where P is a path expression for every $\sigma(c)$ with $c\in\mathcal{C}$. The *restricted selection* on $\mathbf{I}$ with respect to ϕ, denoted $\sigma^r_\phi(\mathbf{I})$, is defined as the TPOB-instance (π',ν') over $\mathbf{S}$, where:

- **$\pi'(c) = \{o \in \pi(c) \mid \sigma^r_\phi(\nu(o))$ is defined$\}$, for all $c \in \mathcal{C}$.**
- **$\nu'(o) = \sigma^r_\phi(\nu(o))$, for all $o \in \pi'(\mathcal{C})$.**

The following example illustrates the use of the restricted selection operator.

Example 11.2.1. Consider the TPOB-instance $\mathbf{I} = (\pi,\nu)$ of Example 11.1.9 and the atomic selection condition $\phi = (\mathsf{Time} < (8,05))$. Then, the restricted selection on $\mathbf{I}$ with respect to ϕ is given by the TPOB-instance $\sigma^r_\phi(\mathbf{I}) = (\pi',\nu')$, where $\pi' = \pi$ and ν' is shown in Table 11.2. □

Table 11.2: ν' resulting from restricted selection

o	$\nu'(o)$
o_3	[Origin: Rome, Destination: Boston, Delivery: $\{((18,00),[.4,.6]),((18,31),[.3,.5])\}$, Time: $\{((8,00),[.45,.5])\}$]

11.3 Renaming

We now define the renaming operation. Informally, this operation renames some attributes in types of TPOB-schemas and in values of TPOB-instances. We use path expressions to allow for a renaming of attributes at lower levels inside types and values. We first define the syntax of renaming conditions, which specify which attributes are to be renamed, and how they are to be renamed.

A *renaming condition* for a probabilistic tuple type τ is an expression of the form $\vec{\mathbf{P}} \leftarrow \vec{\mathbf{Q}}$, where $\vec{\mathbf{P}} = P_1,\ldots,P_l$ is a list of pairwise distinct path expressions for τ, and $\vec{\mathbf{Q}} = Q_1,\ldots,Q_l$ is a list of pairwise distinct path expressions such that P_i and Q_i differ

exactly in their rightmost attribute, for every $i \in \{1, \ldots, l\}$. We illustrate the concept of renaming conditions via our Package Example.

Example 11.3.1. Let τ be the probabilistic tuple type [STOPone: [City: string, Arrive: [[time]], Shipment : [[time]]]]. A renaming condition for τ is STOPone.City, STOPone $\leftarrow$STOPone.City1, STOPone1. □

11.3.1 Renaming of TPOB-Schemas

Before defining how to apply the renaming operator on TPOB-instances, we need two definitions — one on applying it to probabilistic tuple types, and another on applying it to TPOB-schemas.

Let N be a renaming condition of the form $P \leftarrow P'$ for the probabilistic tuple type $\tau = [A_1 : \tau_1, \ldots, A_n : \tau_n]$. The *renaming* of τ with respect to N, denoted $\delta_N(\tau)$, is defined as follows:

- **If $P = A_i$ and $P' = A_i{}'$, then $\delta_N(\tau)$ is obtained from τ by replacing A_i by $A_i{}'$.**
- **If $P = A_i.[[R]]$, $P' = A_i.[[R']]$, and τ_i is an atomic probabilistic type, then $\delta_N(\tau)$ is obtained from τ by replacing $\tau_i = [[\tau_i{}']]$ by $[[\delta_{R \leftarrow R'}(\tau_i{}')]]$.**
- **If $P = A_i.R$, $P' = A_i.R'$, and τ_i is not an atomic probabilistic type, then $\delta_N(\tau)$ is obtained from τ by replacing τ_i by $\delta_{R \leftarrow R'}(\tau_i)$.**

Let $N = P_1, \ldots, P_l \leftarrow P_1{}', \ldots, P_l{}'$ be a renaming condition for τ. The *renaming* of τ with respect to N, denoted $\delta_N(\tau)$, is defined as the simultaneous renaming on τ with respect to all $P_i \leftarrow P_i{}'$.

We now define the renaming of TPOB-schemas as follows. Let $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$ be a TPOB-schema and let N be a renaming condition for every $\sigma(c)$ with $c \in \mathcal{C}$. The *renaming* of S with respect to N, denoted $\delta_N(\mathbf{S})$, is the TPOB-schema $(\mathcal{C}, \sigma', \Rightarrow, \mathrm{me}, \wp)$, where $\sigma'(c) = \delta_N(\sigma(c))$ for all $c \in \mathcal{C}$.

11.3.2 Renaming of TPOB-Instances

Before defining the renaming on TPOB-instances, we need to define it on values of probabilistic tuple types.

Let N be a renaming condition of the form $P \leftarrow P'$ for the probabilistic tuple type $\tau = [A_1 : \tau_1, \ldots, A_n : \tau_n]$. Let $v = [A_1 : v_1, \ldots, A_n : v_n]$ be a value of τ. The *renaming* of v w.r.t. N, denoted $\delta_N(v)$, is defined by:

- **If $P = A_i$ and $P' = A_i'$, then $\delta_N(v)$ is obtained from v by replacing A_i by A_i'.**
- **If $P = A_i.[\![R]\!]$, $P' = A_i.[\![R']\!]$, and v_i is a value of an atomic probabilistic type, then $\delta_N(v)$ is obtained from v by replacing every $(v_i', I_i) \in v_i$ by $(\delta_{R \leftarrow R'}(v_i'), I_i)$.**
- **If $P = A_i.R$, $P' = A_i.R'$, and v_i is not a value of an atomic probabilistic type, then $\delta_N(v)$ is obtained from v by replacing v_i by $\delta_{R \leftarrow R'}(v_i)$.**

Let $N = P_1, \ldots, P_l \leftarrow P_1', \ldots, P_l'$ be a renaming condition for τ. The *renaming* of v with respect to N, denoted $\delta_N(v)$, is defined as the simultaneous renaming on v with respect to all $P_i \leftarrow P_i'$.

We are now ready to define the renaming of TPOB-instances as follows. Let $\mathbf{I} = (\pi, \nu)$ be a TPOB-instance over a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$, and let N be a renaming condition for every $\sigma(c)$ with $c \in \mathcal{C}$. The *renaming* of $\mathbf{I}$ with respect to N, denoted $\delta_N(\mathbf{I})$, is the TPOB-instance (π, ν') over the TPOB-schema $\delta_N(\mathbf{S})$, where $\nu'(o) = \delta_N(\nu(o))$ for all $o \in \pi(\mathcal{C})$.

11.4 Projection

Intuitively, the projection operation removes some top-level attributes (with their associated types) from a TPOB-schema, and the same attributes with their associated values from a TPOB-instance. We formally define the projection of a TPOB-schema (resp., TPOB-instance) on a set of top-level attributes as follows.

The *projection* of a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ on a set of top-level attributes $\mathbf{A}$ of $\mathbf{S}$, denoted $\Pi_{\mathbf{A}}(\mathbf{S})$, is defined as the TPOB-schema $(\mathcal{C}, \sigma', \mathrm{me}, \wp)$, where the new type $\sigma'(c)$ of each class $c \in \mathcal{C}$ is obtained from the old type $\sigma(c) = [B_1 \colon \tau_1, \ldots, B_k \colon \tau_k]$ by removing all $B_i \colon \tau_i$'s with $B_i \notin \mathbf{A}$.

The *projection* of a TPOB-instance $\mathbf{I} = (\pi, \nu)$ over $\mathbf{S}$ on $\mathbf{A}$, denoted $\Pi_{\mathbf{A}}(\mathbf{I})$, is defined as the TPOB-instance (π, ν') over the TPOB-schema $\Pi_{\mathbf{A}}(\mathbf{S})$, where the new value $\nu'(o)$ of each oid $o \in \pi(\mathcal{C})$ is obtained from the old value $\nu(o) = [B_1 \colon v_1, \ldots, B_k \colon v_k]$ by removing all $B_i \colon v_i$'s with $B_i \notin \mathbf{A}$.

The following example illustrates the projection of TPOB-schemas (resp., TPOB - instances).

Example 11.4.1. Consider the fully inherited TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ of Example 10.2.1, the TPOB instance $\mathbf{I} = (\pi, \nu)$ over $\mathbf{S}$ of Example 10.3.1, and the set of attributes

$\mathbf{A} = \{\mathsf{Origin}, \mathsf{Contents}, \mathsf{Time}\}$. The projection of $\mathbf{S}$ (resp., $\mathbf{I}$) on $\mathbf{A}$ results in the TPOB-schema $\Pi_{\mathbf{A}}(\mathbf{S}) = (\mathcal{C}, \sigma', \mathrm{me}, \wp)$ (resp., TPOB-instance $\Pi_{\mathbf{A}}(\mathbf{I}) = (\pi, \nu')$), where σ' (resp., ν') is shown in Table 11.3 (resp., 11.4). □

Table 11.3: σ' resulting from projection

c	$\sigma'(c)$
Package	[Origin: string]
Letter	[Origin: string]
Box	[Origin: string, Contents: {string}]
Tube	[Origin: string, Contents: {string}]
Priority	[Origin: string, Time: [[time]]]
Express_saves	[Origin: string, Time: [[time]]]
One-transfer	[Origin: string, Contents: {string}, Time: [[time]]]
Two-transfer	[Origin: string, Contents: {string}, Time: [[time]]]

Table 11.4: ν' resulting from projection

o	$\nu'(o)$
o_3	[Origin: Rome, Time: $\{((8,00),[.45,.5]),((8,10),[.4,.5])\}$]
o_5	[Origin: Paris, Contents: {photos, books}, Time: $\{((12,00),[.3,.4]),((12,05),[.4,.7])\}$]

11.5 Extraction

The extraction operation allows for the elimination of some classes from the class hierarchy of a TPOB-schema. That is, the extraction operation removes some classes from a TPOB-schema, and all objects in the removed classes from a TPOB-instance. This is often useful when we wish to focus interest on a certain part of the TPOB-schema (e.g. the part consisting of letters and its subclasses only) and/or TPOB-instances. We first define extraction operation on TPOB-schemas and then extend it to TPOB-instances.

Extraction on TPOB-Schemas

Roughly speaking, the extraction operation on a TPOB-schema can be described as follows. Given a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ and a set of classes $\mathbf{C} \subseteq \mathcal{C}$, the extraction on $\mathbf{S}$ with respect to $\mathbf{C}$ produces the TPOB schema $\mathbf{S}' = (\mathbf{C}, \sigma', \mathrm{me}', \wp')$ that

is obtained by restricting $\mathbf{S}$ to the classes in $\mathbf{C}$. That is, the new TPOB-schema $\mathbf{S}'$ contains only the classes in $\mathbf{C}$ and their types. Moreover, the mapping me and the probability assignment $\wp$ are adapted to the classes in $\mathbf{C}$ in order to obtain me' and $\wp'$. For example, if we have the immediate subclass relationships $c_1 \Rightarrow c_2 \Rightarrow c_3$ in $\mathbf{S}$ with the associated probabilities $\wp(c_1, c_2)$ and $\wp(c_2, c_3)$, and c_1 and c_3 belong to $\mathbf{C}$ but c_2 does not, then we assume the immediate subclass relationship $c_1 \Rightarrow' c_3$ in $\mathbf{S}'$ with the associated probability $\wp'(c_1, c_3) = \wp(c_1, c_2) \cdot \wp(c_2, c_3)$. Moreover, if a class $c_1 \in \mathbf{C}$ is disjoint to a class $c_2 \notin \mathbf{C}$ in $\mathbf{S}$, then all the extracted subclasses of c_2 are disjoint to c_1 in $\mathbf{S}'$.

More formally, the *extraction* on a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ with respect to a set of classes $\mathbf{C} \subseteq \mathcal{C}$, denoted $\Xi_{\mathbf{C}}(\mathbf{S})$, is the TPOB-schema $\mathbf{S}' = (\mathcal{C}', \sigma', \mathrm{me}', \wp')$, where:

- $\mathcal{C}' = \mathbf{C}$.
- σ' **is the restriction of** σ **to** $\mathcal{C}'$.
- **For all** $c \in \mathcal{C}'$, **we define** $\mathrm{me}'(c) = \{((X \cap \mathcal{C}') \cup X_{\mathcal{C}'}) \neq \emptyset \mid X \in \mathrm{me}(c)\}$, **where** $X_{\mathcal{C}'} = \{d_1 \in \mathcal{C}' \mid d_1 \Rightarrow d_2 \Rightarrow \cdots \Rightarrow d_k$ **for some** $d_2, \ldots, d_{k-1} \in \mathcal{C} - \mathcal{C}'$ **and** $d_k \in X - \mathcal{C}'\}$, **and we assume that** $|X_{\mathcal{C}'}| \leq 1$.
- **For all** $c_1, c_2 \in \mathcal{C}'$, **we define** $\wp'(c_1, c_2) = \prod_{i=1}^{k-1} \wp(d_i, d_{i+1})$, **where the** d_i**'s are such that** $d_1 \Rightarrow d_2 \Rightarrow \cdots \Rightarrow d_k$, $d_1 = c_1$, $d_k = c_2$, **and** $d_2, \ldots, d_{k-1} \in \mathcal{C} - \mathcal{C}'$.

The following example illustrates the use of the extraction operator on TPOB-schemas.

Example 11.5.1. Consider the fully inherited TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ of Example 10.2.1. The extraction on $\mathbf{S}$ w.r.t. the set of classes $\mathbf{C} = \{\mathsf{Package}, \mathsf{Letter}, \mathsf{Priority}, \mathsf{One\text{-}transfer}\}$ is given by the TPOB-schema $\Xi_{\mathbf{C}}(\mathbf{S}) = (\mathcal{C}', \sigma', \mathrm{me}', \wp')$, where:

- $\mathcal{C}' = \{\mathsf{Package}, \mathsf{Letter}, \mathsf{Priority}, \mathsf{One\text{-}transfer}\}$.
- The type assignment σ' is given in Table 11.5.
- The probability assignment $\wp'$ is given in Figure 11.1.
- $\mathrm{me}'(\mathsf{Package}) = \{\{\mathsf{Letter}, \mathsf{One\text{-}transfer}\}, \{\mathsf{Priority}\}\}$, $\mathrm{me}'(\mathsf{Priority}) = \{\{\mathsf{One\text{-}transfer}\}\}$, and $\mathrm{me}'(\mathsf{Letter}) = \mathrm{me}'$ $(\mathsf{One\text{-}transfer}) = \emptyset$. □

Table 11.5: Type assignment σ' resulting from extraction

c	$\sigma'(c)$
Package	[Origin : string, Destination : string, Delivery : [[time]]]
Letter	[Origin : string, Destination : string, Delivery : [[time]], Height : float, Width : float]
Priority	[Origin : string, Destination : string, Delivery : [[time]], Time : [[time]]]
One-transfer	[Origin : string, Destination : string, Delivery : [[time]], Height : float, Width : float, Depth : float, Contents : {string}, Time : [[time]], City : string, Arrive : [[time]], Shipment : [[time]]]

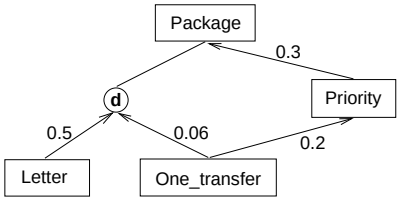


Figure 11.1: Class hierarchy and probability assignment resulting from extraction

11.5.1 Extraction on TPOB-instances

The extraction on a TPOB-instance I with respect to a set of classes C returns the TPOB-instance I' that contains all the objects (and their values) in I that belong to the classes in C. Formally, the *extraction* on a TPOB-instance $\mathrm{I} = (\pi, \nu)$ over a TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ with respect to a set of classes $\mathrm{C} \subseteq \mathcal{C}$, denoted $\Xi_{\mathbf{C}}(\mathbf{I})$, is the TPOB-instance (π', ν') over the TPOB-schema $\Xi_{\mathbf{C}}(\mathbf{S}) = (\mathcal{C}', \sigma', \mathrm{me}', \wp')$, where:

- **π' is the restriction of π to $\mathcal{C}'$.**
- **ν' is the restriction of ν to $\pi'(\mathcal{C}')$.**

The following example illustrates the extraction on TPOB-instances.

Example 11.5.2. Consider the fully inherited TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ of Example 10.2.1 and the TPOB-instance $\mathbf{I} = (\pi, \nu)$ over $\mathbf{S}$ of Example 10.3.1. The extraction on $\mathbf{I}$ with respect to the set of classes $\mathbf{C} = \{\mathsf{PackageLetter}, \mathsf{Priority}, \mathsf{One\text{-}transfer}\}$ results in the TPOB-instance $\Xi_{\mathbf{C}}(\mathbf{I}) = (\pi', \nu')$ over the TPOB-schema $\Xi_{\mathbf{C}}(\mathbf{S})$ in Example 11.5.1, where π' is given by Table 11.6 and ν' is given by $\nu'(o_3) = \nu(o_3)$. □

Table 11.6: π' and $(\pi')^*$ resulting from extraction

c	$\pi'(c)$	$(\pi')^*(c)$
Package	$\{\}$	$\{o_3\}$
Letter	$\{\}$	$\{\}$
Priority	$\{o_3\}$	$\{o_3\}$
One-transfer	$\{\}$	$\{\}$

Chapter 12

TPOB-Algebra: Binary Operations

In this Chapter, we define algebraic operations to combine information from two TPOB-instances. These operations take two TPOB-instances over two TPOB-schemas as input and produce a new TPOB-instance over a (possibly new) TPOB-schema as output. We consider the binary operations of natural join, Cartesian product, conditional join, and the set operators of intersection, union, and difference. Again, unless specified otherwise, we assume that all input TPOB-schemas are fully inherited.

12.1 Natural Join

Our *natural join* operator is inspired by its counterpart in classical relational databases, which forms a Cartesian product of two TPOB-instances, performs a selection forcing equality on those attributes that appear in both relation schemes, and finally removes duplicate values.

The main idea behind the natural join operation can be roughly described as follows. Recall that for each class $c \in \mathcal{C}$ of a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$, the type $\sigma(c) = [A_1 : \tau_1, \ldots, A_l : \tau_l]$ is a probabilistic tuple type over some top-level attributes $A_1, \ldots, A_l$. Moreover, each object $o \in \pi(c)$ in a TPOB-instance $\mathbf{I} = (\pi, \nu)$ over $\mathbf{S}$ is associated with a value $\nu(o) = [A_1 : v_1, \ldots, A_l : v_l]$ of type $\sigma(c)$. Given two TPOB-instances $\mathbf{I}_1$ and $\mathbf{I}_2$ over the TPOB-schemas $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively, the TPOB-instance $\mathbf{I}$ resulting from the natural join of $\mathbf{I}_1$ and $\mathbf{I}_2$ contains an object $o = (o_1, o_2)$ for certain pairs of objects o_1 and o_2 in $\mathbf{I}_1$ and $\mathbf{I}_2$, respectively. The value $\nu(o)$ associated with each such o is given by the natural join of the values $\nu_1(o_1)$ and $\nu_2(o_2)$ in $\mathbf{I}_1$ and $\mathbf{I}_2$,

respectively, which is roughly a concatenation combined with an intersection on common attribute values. The new TPOB-instance I is defined over a TPOB-schema S that contains a class $c = (c_1, c_2)$ for any two classes c_1 and c_2 in S_1 and S_2, respectively. The type of each such class c is obtained by merging the types of c_1 and c_2 in S_1 and S_2, respectively, and the new class hierarchy in S is obtained by merging the two class hierarchies in S_1 and S_2.

We first formalize the natural join of two TPOB-schemas. We then define the natural join of two values of probabilistic tuple types. Finally, we introduce the natural join of two TPOB-instances. In the rest of this section, let $\mathrm{S}_1 = (\mathcal{C}_1, \sigma_1, \mathrm{me}_1, \wp_1)$ and $\mathrm{S}_2 = (\mathcal{C}_2, \sigma_2, \mathrm{me}_2, \wp_2)$ be two TPOB-schemas.

12.1.1 Natural Join of TPOB-Schemas

Informally, the TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ produced by the natural join of the TPOB-schemas S_1 and S_2 is obtained as follows. First, the set of classes $\mathcal{C}$ is the Cartesian product of the sets of classes $\mathcal{C}_1$ and $\mathcal{C}_2$. Second, the type $\sigma(c)$ of each class $c = (c_1, c_2) \in \mathcal{C}$ is given by the probabilistic tuple type containing every top-level attribute with its associated type in $\sigma_1(c_1)$ and $\sigma_2(c_2)$. Note that this assumes that every common top-level attribute of $\sigma_1(c_1)$ and $\sigma_2(c_2)$ has the same type in $\sigma_1(c_1)$ and $\sigma_2(c_2)$. We say S_1 and S_2 are *natural-join-compatible* when this condition is satisfied for any two classes $c_1 \in \mathcal{C}_1$ and $c_2 \in \mathcal{C}_2$. Third, the class hierarchy in S is defined as the Cartesian product of the class hierarchies in S_1 and S_2. That is, every class $c_1 \in \mathcal{C}_1$ (resp., $c_2 \in \mathcal{C}_2$) is combined with the immediate subclass and disjointness relationships expressed by every $\mathrm{me}_2(c_2)$ with $c_2 \in \mathcal{C}_2$ (resp., $\mathrm{me}_1(c_1)$ with $c_1 \in \mathcal{C}_1$); every class $c_1 \in \mathcal{C}_1$ (resp., $c_2 \in \mathcal{C}_2$) is combined with every conditional probability $\wp_2(c_2, d_2)$ in S_2 (resp., $\wp_1(c_1, d_1)$ in S_1).

More formally, we define the natural join operation on two TPOB-schemas as follows. If the two TPOB-schemas S_1 and S_2 are natural-join-compatible, then the *natural join* of S_1 and S_2, denoted $\mathrm{S}_1 \bowtie \mathrm{S}_2$, is defined as the TPOB-schema $\mathrm{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$, where

- **$\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2$.**
- **For each $c = (c_1, c_2) \in \mathcal{C}$, the probabilistic tuple type $\sigma(c) = [A_1 : \tau_1, \ldots, A_l : \tau_l]$ contains exactly every $A_i : \tau_i$ that belongs to either the type $\sigma_1(c_1)$ or the type $\sigma_2(c_2)$.**

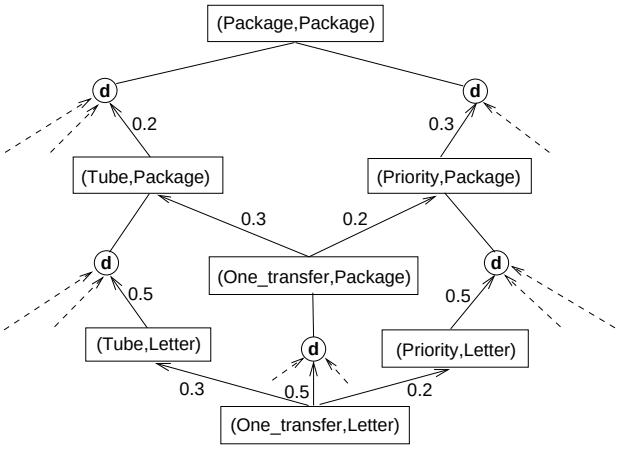


Figure 12.1: Natural join of schemas

- **For each** $c{=}(c_1, c_2){\in}\mathcal{C}$**:** $\mathrm{me}(c) = \{\{c_1\}{\times}\mathcal{P}_2 \mid \mathcal{P}_2 \in \mathrm{me}_2(c_2)\} \cup \{\mathcal{P}_1{\times}\{c_2\} \mid \mathcal{P}_1 \in \mathrm{me}_1(c_1)\}$**.**
- **For each immediate subclass relationship** $(c_1, c_2) \Rightarrow (c_1, d_2)$ **(resp.,** $(c_1, c_2) \Rightarrow (d_1, c_2)$**) in S, define** $\wp((c_1, c_2), (c_1, d_2)) = \wp_2(c_2, d_2)$ **(resp.,** $\wp((c_1, c_2), (d_1, c_2)) = \wp_1(c_1, d_1)$**).**

The following example illustrates the natural join of TPOB-schemas via the Package Example.

Example 12.1.1. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be the TPOB-schemas of Examples 10.2.1 and 11.5.1, respectively. Their natural join $\mathbf{S}_1 \bowtie \mathbf{S}_2$ is the TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ partially shown in Table 12.1 and Figure 12.1. □

Table 12.1: Type assignment σ resulting from natural join

c	$\sigma(c)$
(Package, Package)	[Origin: string, Destination: string, Delivery: [[time]]]
(Tube, Package)	[Origin: string, Destination: string, Delivery: [[time]], Height: float, Width: float, Depth: float, Contents: {string}]
(Priority, Package)	[Origin: string, Destination: string, Delivery: [[time]], Time: [[time]]]
(One-transfer, Letter)	[Origin: string, Destination: string, Delivery: [[time]], Height: float, Width: float, Depth: float, Contents: {string}, Time: [[time]], City: string, Arrive: [[time]], Shipment: [[time]]]

12.1.2 Intersection and Natural Join of Values

The natural join of two classical relations R and S contains one tuple for each pair of tuples $(r, s) \in R \times S$ that have the same values in the common attributes of R and S. Similarly, the natural join of two TPOB-instances $\mathbf{I}_1$=(π_1, ν_1) and $\mathbf{I}_2$=(π_2, ν_2) contains one object o for each pair of objects o_1 in $\mathbf{I}_1$ and o_2 in $\mathbf{I}_2$ such that the natural join of $\nu_1(o_1)$ and $\nu_2(o_2)$ (their concatenation combined with the intersection of the values in the common attributes) is defined, which then forms the value $\nu(o)$ of the new object o.

To define the natural join of two values v_1 and v_2 of probabilistic tuple types τ_1 and τ_2, respectively, we now first introduce the intersection of two values v_1 and v_2 of the same classical or probabilistic type τ. The *intersection* of v_1 and v_2 under a conjunction strategy $\otimes$, denoted $v_1 \cap_\otimes v_2$, is inductively defined by:

- **If τ is a classical type and $v_1 = v_2$, then $v_1 \cap_\otimes v_2 = v_1$.**
- **If τ is an atomic probabilistic type and $w \neq \emptyset$, then $v_1 \cap_\otimes v_2 = w$, where**

$$w = \{(v, I_1 \otimes I_2) \mid (v, I_1) \in v_1,\ (v, I_2) \in v_2\}\,.$$

- **If τ is a probabilistic tuple type over the set of top-level attributes $\mathbf{A}$ and all $v_1.A \cap_\otimes v_2.A$ are defined, then $(v_1 \cap_\otimes v_2).A = v_1.A \cap_\otimes v_2.A$ for all $A \in \mathbf{A}$.**

Otherwise, $v_1 \cap_\otimes v_2$ is undefined. The following example illustrates the above concept.

Example 12.1.2. Let time be the standard calendar w.r.t. $hour \sqsupseteq minute$, and let $\otimes$ be a conjunction strategy. Consider the values v_1=v_2=$(12, 30)$ and v_3=$(12, 40)$ of the classical type time. Then, $v_1 \cap_\otimes v_2 = v_1$, while $v_1 \cap_\otimes v_3$ is undefined. Now consider the following values of the probabilistic type $[\![\mathsf{time}]\!]$:

$$v_1 = \{((9,00),[.3,.5]),((10,00),[.3,.6]),((11,00),[.2,.5])\},\ v_2 = \{((12,00),[.2,.4])\}\,,$$
$$v_3 = \{((9,00),[.3,.4]),((11,00),[.3,.6]),((12,00),[.2,.4])\}\,.$$

Then, $v_1 \cap_{\otimes_{in}} v_2$ is undefined, while $v_1 \cap_{\otimes_{in}} v_3 = \{((9,00),[.09,.2]),((11,00),[.06,.3])\}$. □

We are now ready to define the natural join of two values v_1 and v_2 of probabilistic tuple types τ_1 and τ_2, respectively. Let $\otimes$ be a conjunction strategy. Let $\mathbf{A}_1$ and

$\mathbf{A}_2$ be the top-level attributes of τ_1 and τ_2, respectively, and let $\mathbf{A} = \mathbf{A}_1 \cap \mathbf{A}_2$. Let all $A \in \mathbf{A}$ have the same types in τ_1 and τ_2. The *natural join* of v_1 and v_2 under $\otimes$, denoted $v_1 \bowtie_\otimes v_2$, is defined as follows:

- **$(v_1 \bowtie_\otimes v_2).A = v_i.A$ for all $A \in \mathbf{A}_i - \mathbf{A}$, $i \in \{1,2\}$, $(v_1 \bowtie_\otimes v_2).A = v_1.A \bowtie_\otimes v_2.A$ for all $A \in \mathbf{A}$. If all $v_1.A \bowtie_\otimes v_2.A$ with $A \in \mathbf{A}$ are defined, then $v_1 \bowtie_\otimes v_2$ is defined.**

12.1.3 Natural Join of TPOB-Instances

We now define the natural join of two TPOB-instances as follows. Let $\mathbf{I}_1 = (\pi_1, \nu_1)$ and $\mathbf{I}_2 = (\pi_2, \nu_2)$ be two TPOB-instances over the natural-join-compatible TPOB-schemas $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively. For $i \in \{1,2\}$, let $\mathbf{A}_i$ denote the set of top-level attributes of $\mathbf{S}_i$. Let $\otimes$ be a conjunction strategy. The *natural join* of $\mathbf{I}_1$ and $\mathbf{I}_2$ under $\otimes$, denoted $\mathbf{I}_1 \bowtie_\otimes \mathbf{I}_2$, is the TPOB-instance (π, ν) over $\mathbf{S}_1 \bowtie \mathbf{S}_2$, where:

- **$\pi(c) = \{(o_1, o_2) \in \pi_1(c_1) \times \pi_2(c_2) \mid \nu_1(o_1) \bowtie_\otimes \nu_2(o_2)$ is defined$\}$, for all $c = (c_1, c_2) \in \mathcal{C}_1 \times \mathcal{C}_2$.**
- **$\nu(o) = \nu_1(o_1) \bowtie_\otimes \nu_2(o_2)$, for all $o = (o_1, o_2) \in \pi(\mathcal{C}_1 \times \mathcal{C}_2)$.**

Example 12.1.3. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be the TPOB-schemas given in Example 10.2.1 and produced in Example 11.5.1, respectively. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be the TPOB-instances over $\mathbf{S}_1$ and $\mathbf{S}_2$ produced in Examples 11.1.9 and 11.5.2, respectively. The natural join of $\mathbf{I}_1$ and $\mathbf{I}_2$ under the conjunction strategy for ignorance $\otimes_{ig}$ is the TPOB-instance $\mathbf{I}_1 \bowtie_{\otimes_{ig}} \mathbf{I}_2 = (\pi, \nu)$ over $\mathbf{S}_1 \bowtie \mathbf{S}_2 = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$, where π is given by $\pi((\mathsf{Priority}, \mathsf{Priority})) = \{(o_3, o_3)\}$ and $\pi(c) = \emptyset$ for all other classes $c \in \mathcal{C}$, and ν is given by Table 12.2. □

Table 12.2: ν resulting from natural join

o	$\nu(o)$
(o_3, o_3)	[Origin: Rome, Destination: Boston, Delivery: $\{((18,00),[.4,1]),((18,31),[.3,1])\}$, Time: $\{((8,00),[.45,1]),((8,10),[.4,1])\}$]

12.2 Cartesian Product and Conditional Join

In the above definition of natural join, if the sets $\mathbf{A_1}$ and $\mathbf{A_2}$ are disjoint, then the natural join is called *Cartesian product* and denoted by the symbol $\times$. Two TPOB-schemas S_1 and S_2 are *Cartesian-product-compatible* iff they can be combined using Cartesian product, that is, iff for all classes c_1 in S_1 and c_2 in S_2, the types of c_1 and c_2 have disjoint sets of top-level attributes.

The conditional join operation combines values of two TPOB-instances that satisfy a probabilistic selection condition ϕ. Let I_1 and I_2 be TPOB-instances over the Cartesian-product-compatible TPOB-schemas S_1 and S_2, respectively. The *conditional join* of I_1 and I_2 with respect to ϕ, denoted $\mathrm{I}_1 \bowtie_\phi \mathrm{I}_2$, is the TPOB-instance $\sigma_\phi(\mathbf{I}_1 \times \mathbf{I}_2)$ over the TPOB-schema $\mathrm{S}_1 \times \mathrm{S}_2$.

Example 12.2.1. Let $\mathbf{S}_1$ and $\mathbf{I}_1$ be the TPOB-schema and the TPOB-instance, respectively, produced in Example 11.4.1. Let $\mathbf{S}_2$ and $\mathbf{I}_2$ be the TPOB-schema and the TPOB-instance obtained from $\mathbf{S}_1$ and $\mathbf{I}_1$, respectively, by renaming the attributes Origin, Contents, and Time with Origin1, Contents1, and Time1, respectively. The Cartesian product of $\mathbf{S}_1$ and $\mathbf{S}_2$ is the TPOB-schema $\mathbf{S}_1 \times \mathbf{S}_2 = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ partially shown in Table 12.3 and Figure 12.1. The condition join of $\mathbf{I}_1$ and $\mathbf{I}_2$ with respect to $\phi = (\mathsf{Origin} = \mathsf{Rome} \wedge \mathsf{Origin1} = \mathsf{Paris})[1, 1]$ is the TPOB-instance $\sigma_\phi(\mathbf{I}_1 \times \mathbf{I}_2) = (\pi, \nu)$ over the TPOB-schema $\mathbf{S}_1 \times \mathbf{S}_2$, where $\pi((\mathsf{Priority}, \mathsf{Two\text{-}transfer})) = \{(o_3, o_5)\}$ and $\pi(c) = \emptyset$ for all other $c \in \mathcal{C}$, and ν is shown in Table 12.4. □

Table 12.3: Type assignment σ resulting from conditional join

c	$\sigma(c)$
(Package, Package)	[Origin: string, Origin1: string]
(Tube, Package)	[Origin: string, Contents: {string}, Origin1: string]
(Priority, Package)	[Origin: string, Time: [[time]], Origin1: string]
(One-transfer, Letter)	[Origin: string, Contents: {string}, Time: [[time]], Origin1: string]

Table 12.4: Value assignment ν resulting from conditional join

o	$\nu(o)$
(o_3, o_5)	[Origin: Rome, Time: $\{((8,00),[.45,.5]),((8,10),[.4,.5])\}$, Origin1: Paris, Time1: $\{((12,00),[.3,.4]),((12,05),[.4,.7])\}$]

12.3 Intersection, Union, and Difference

In this section, we define the set operations of intersection, union, and difference on two TPOB-instances over the same TPOB-schema. Informally, the intersection operation intersects the sets of objects of the two TPOB instances, as well as the two values associated with each object in both TPOB-instances. The union operation, in contrast, computes the union of the sets of objects of the two TPOB-instances, combined with the union of the two values associated with each object in both TPOB-instances. Finally, the difference operation returns the set of objects of the first TPOB-instance, combined with the difference of the two values associated with each object in both TPOB-instances.

We first define the intersection of two TPOB-instances. We then introduce the union of two values and two TPOB-instances, and finally the difference of two values and two TPOB-instances.

12.3.1 Intersection of TPOB-Instances

We formally define the intersection of two TPOB-instances as follows. Let $\mathbf{I}_1$=(π_1, ν_1) and $\mathbf{I}_2$=(π_2, ν_2) be TPOB-instances over the same TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$, and let $\otimes$ be a conjunction strategy. Then *intersection* of $\mathbf{I}_1$ and $\mathbf{I}_2$ under $\otimes$, denoted $\mathbf{I}_1 \cap_\otimes \mathbf{I}_2$, is the TPOB-instance (π, ν) over $\mathbf{S}$, where:

- $\pi(c) = \{o \in \pi_1(c) \cap \pi_2(c) \mid \nu_1(o) \cap_\otimes \nu_2(o)$ **is defined** $\}$**, for all** $c \in \mathcal{C}$**.**
- $\nu(o) = \nu_1(o) \cap_\otimes \nu_2(o)$**, for all** $o \in \pi(\mathcal{C})$**.**

The following example illustrates the intersection operation on two TPOB-instances.

Example 12.3.1. Let $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ be the TPOB-schema of Example 10.2.1. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be the TPOB instances over $\mathbf{S}$ given in Example 10.3.1 and produced in Example 11.2.1, respectively. Then, the intersection of $\mathbf{I}_1$ and $\mathbf{I}_2$ under the conjunction strategy for ignorance is the TPOB-instance $\mathbf{I}_1 \cap_{\otimes_{ig}} \mathbf{I}_2 = (\pi, \nu)$ over $\mathbf{S}$, where π is given by $\pi(\mathsf{Priority}) = \{o_3\}$ and $\pi(c) = \emptyset$ for all other $c \in \mathcal{C}$, and ν is shown in Table 12.5. □

Table 12.5: ν resulting from intersection

o	$\nu(o)$
o_3	[Origin: Rome, Destination: Boston, Delivery: $\{((18,00),[.4,1]),((18,31),[.3,1])\}$, Time: $\{((8,00),[.45,1])\}$]

12.3.2 Union of Values

We now define the union of two values. Let v_1 and v_2 be two values of the same classical or probabilistic type τ. The *union* of v_1 and v_2 under a disjunction strategy $\oplus$, denoted $v_1 \cup_\oplus v_2$, is inductively defined by:

- **If τ is a classical type and $v_1 = v_2$, then $v_1 \cup_\oplus v_2 = v_1$.**
- **If τ is an atomic probabilistic type, then**

$$\begin{aligned} v_1 \cup_\oplus v_2 \;\; = \;\; & \{(v, I_1) \in v_1 \mid v \in V_1 - V_2\} \cup \{(v, I_2) \in v_2 \mid v \in V_2 - V_1\} \cup \\ & \{(v, I_1 \oplus I_2) \mid (v, I_1) \in v_1,\ (v, I_2) \in v_2\}\,, \end{aligned}$$

  where $V_1 = \{v \mid (v, I) \in v_1\}$ and $V_2 = \{v \mid (v, I) \in v_2\}$.

- **If τ is a probabilistic tuple type over the set of top-level attributes A and all $v_1.A \cup_\oplus v_2.A$ are defined, then $(v_1 \cup_\oplus v_2).A = v_1.A \cup_\oplus v_2.A$ for all $A \in$ A.**

Otherwise, $v_1 \cup_\oplus v_2$ is undefined. The following example illustrates the above concept.

Example 12.3.2. Let time be the standard calendar w.r.t. $hour \sqsupseteq minute$, and let $\oplus$ be a disjunction strategy. Consider the values v_1=v_2=$(12, 30)$ and v_3=$(12, 40)$ of the classical type time. Then, $v_1 \cup_\oplus v_2 = v_1$, while $v_1 \cup_\oplus v_3$ is undefined. Consider next the following values of the probabilistic type $[\![\text{time}]\!]$:

$$v_1 = \{((9,00),[.3,.5]),((10,00),[.3,.6]),((11,00),[.2,.5])\},\ v_2 = \{((12,00),[.2,.4])\}\,,$$
$$v_3 = \{((9,00),[.3,.4]),((11,00),[.3,.6]),((12,00),[.2,.4])\}\,.$$

Then, we have $v_1 \cup_{\oplus_{in}} v_2 = \{((9,00),[.3,.5]),((10,00),[.3,.6]),((11,00),[.2,.5]),((12,00),[.2,.4])\}$, while $v_1 \cup_{\oplus_{in}} v_3 = \{((9,00),[.51,.7]),((10,00),[.3,.6]),((11,00),[.44,.8]),((12,00),[.2,.4])\}$. □

12.3.3 Union of TPOB-Instances

We are now ready to define the union of two TPOB-instances. Let $\mathbf{I}_1 = (\pi_1, \nu_1)$ and $\mathbf{I}_2 = (\pi_2, \nu_2)$ be two TPOB-instances over the same TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$, and let $\oplus$ be a disjunction strategy. The *union* of $\mathbf{I}_1$ and $\mathbf{I}_2$ under $\oplus$, denoted $\mathbf{I}_1 \cup_\oplus \mathbf{I}_2$, is the TPOB-instance (π, ν) over $\mathbf{S}$, where:

- $\pi(c) = (\pi_1(c) - \pi_2(c)) \cup (\pi_2(c) - \pi_1(c)) \cup \{o \in \pi_1(c) \cap \pi_2(c) \mid \nu_1(o) \cup_\oplus \nu_2(o)$ **is defined**$\}$, **for all** $c \in \mathcal{C}$.
- $\nu(o) = \begin{cases} \nu_1(o) & \textbf{if } o \in \pi_1(c) - \pi_2(c) \\ \nu_2(o) & \textbf{if } o \in \pi_2(c) - \pi_1(c) \\ \nu_1(o) \cup_\oplus \nu_2(o) & \textbf{if } o \in \pi_1(c) \cap \pi_2(c) \,. \end{cases}$

The following example illustrates the union operation on two TPOB-instances.

Example 12.3.3. Let $\mathbf{S}$ be the TPOB-schema of Example 10.2.1. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be the TPOB-instances over $\mathbf{S}$ given in Example 10.3.1 and produced in Example 11.2.1, respectively. Then, the union of $\mathbf{I}_1$ and $\mathbf{I}_2$ under the disjunction strategy for positive correlation is the TPOB-instance $\mathbf{I}_1 \cup_{\oplus_{pc}} \mathbf{I}_2 = \mathbf{I}_1$. □

12.3.4 Difference of Values

We now define the difference of values. Let v_1 and v_2 be values of the same classical or probabilistic type τ. The *difference* of v_1 and v_2 under a difference strategy $\ominus$, denoted $v_1 -_\ominus v_2$, is inductively defined by:

- **If τ is a classical type and $v_1 = v_2$, then $v_1 -_\ominus v_2 = v_1$.**
- **If τ is an atomic probabilistic type, then**

$$v_1 -_\ominus v_2 \;=\; \{(v, I_1) \in v_1 \mid v \in V_1 - V_2\} \cup \{(v, I_1 \ominus I_2) \mid (v, I_1) \in v_1,\ (v, I_2) \in v_2\}\,,$$

  where $V_1 = \{v \mid (v, I) \in v_1\}$ and $V_2 = \{v \mid (v, I) \in v_2\}$.
- **If τ is a probabilistic tuple type over the set of top-level attributes $\mathbf{A}$ and all $v_1.A -_\ominus v_2.A$ are defined, then $(v_1 -_\ominus v_2).A = v_1.A -_\ominus v_2.A$ for all $A \in \mathbf{A}$.**

Otherwise, $v_1 -_\ominus v_2$ is undefined.

12.3.5 Difference of TPOB-Instances

We now define the difference of two TPOB-instances. Let $\mathbf{I}_1 = (\pi_1, \nu_1)$ and $\mathbf{I}_2 = (\pi_2, \nu_2)$ be TPOB-instances over the same TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$, and let $\ominus$ be a difference strategy. The *difference* of $\mathbf{I}_1$ and $\mathbf{I}_2$ under $\ominus$, denoted $\mathbf{I}_1 -_\ominus \mathbf{I}_2$, is the TPOB-instance (π, ν) over S, where:

- $\pi(c) = (\pi_1(c) - \pi_2(c)) \cup \{o \in \pi_1(c) \cap \pi_2(c) \mid \nu_1(o) -_\ominus \nu_2(o)$ **is defined** $\}$**, for all** $c \in \mathcal{C}$**.**
- $\nu(o) = \begin{cases} \nu_1(o) & \textbf{if } o \in \pi_1(c) - \pi_2(c) \\ \nu_1(o) -_\ominus \nu_2(o) & \textbf{if } o \in \pi_1(c) \cap \pi_2(c) \textbf{.} \end{cases}$

Chapter 13

Preservation of Consistency and Coherence

We now show that all algebraic operators defined in Chapters 11 and 12 preserve consistency and coherence of schemas and instances. If the input TPOB-schemas (resp., TPOB-instances) are consistent (resp., coherent), then the output TPOB-schemas (resp., TPOB-instances) are also consistent (resp., coherent).

The operators of selection, restricted selection, intersection, union, and difference trivially preserve consistency of schemas, as the input schemas coincide with the output schemas. Renaming and projection also preserve consistency of schemas, as they only modify type assignments. The following result shows that extraction and natural join, and thus also Cartesian product and conditional join, preserve consistency of schemas.

Theorem 13.0.4. *Let* S *be a TPOB-schema, and let* C *be a set of classes from* S*. Let* S_1 *and* S_2 *be two natural-join-compatible TPOB-schemas.*

(a) If $\mathbf{S}$ *is consistent, then* $\Xi_{\mathbf{C}}(\mathbf{S})$ *is consistent.*

(b) If $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *are consistent, then* $\mathbf{S}_1 \bowtie \mathbf{S}_2$ *is consistent.*

Proof of Theorem 13.0.4. (a) Let $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$**, and let** $\Xi_{\mathbf{C}}(\mathbf{S}) = \mathbf{S}' = (\mathcal{C}', \sigma', \Rightarrow', \mathrm{me}', \wp')$**. Assume that S is consistent. That is, there exists a model** $\varepsilon \colon \mathcal{C} \to 2^{\mathcal{O}}$ **of S. Let the mapping** $\varepsilon' \colon \mathcal{C}' \to 2^{\mathcal{O}}$ **be defined by** $\varepsilon'(c) = \varepsilon(c)$ **for all** $c \in \mathcal{C}'$**. We now show that** ε' **is a model of S′. C1 holds, as** $\varepsilon(c) \neq \emptyset$ **for all classes** $c \in \mathcal{C}$ **implies** $\varepsilon'(c) \neq \emptyset$ **for all classes** $c \in \mathcal{C}'$**. We next show C2. Consider two classes** $c_1, c_2 \in \mathcal{C}'$

such that $c_1 \Rightarrow' c_2$. That is, some path $d_1 \Rightarrow d_2 \Rightarrow \cdots \Rightarrow d_k$ exists such that $d_1 = c_1$, $d_k = c_2$, and $d_2, \ldots, d_{k-1} \in \mathcal{C} - \mathcal{C}'$. As $\varepsilon(d_1) \subseteq \varepsilon(d_2) \subseteq \cdots \subseteq \varepsilon(d_k)$, it thus follows $\varepsilon(c_1) \subseteq \varepsilon(c_2)$. We now prove that C3 holds. Let $c_1, c_2 \in \mathcal{C}'$ be two distinct classes that belong to the same cluster $\mathcal{P}' \in \mathrm{me}'(c)$ for some $c \in \mathcal{C}'$. That is, there exists a cluster $\mathcal{P} \in \mathrm{me}(c)$ such that, for $i \in \{1, 2\}$, either c_i belongs to $\mathcal{P}$ or c_i is a proper subclass of a class in $\mathcal{P}$. As C2 and C3 hold for ε, it thus follows that $\varepsilon(c_1) \cap \varepsilon(c_2) = \emptyset$. This shows that C3 holds. We finally prove C4. Consider two classes $c_1, c_2 \in \mathcal{C}'$ such that $c_1 \Rightarrow' c_2$. That is, some path $d_1 \Rightarrow d_2 \Rightarrow \cdots \Rightarrow d_k$ exists such that $d_1 = c_1$, $d_k = c_2$, and $d_2, \ldots, d_{k-1} \in \mathcal{C} - \mathcal{C}'$. Moreover, it holds $\wp'(c_1, c_2) = \prod_{i=1}^{k-1} \wp(d_i, d_{i+1})$. As C4 holds for ε, it follows that $|\varepsilon(d_i)| = \wp(d_i, d_{i+1}) \cdot |\varepsilon(d_{i+1})|$ for all $i \in \{1, \ldots, k-1\}$. This shows that $|\varepsilon(c_1)| = \prod_{i=1}^{k-1} \wp(d_i, d_{i+1}) \cdot |\varepsilon(c_2)|$, that is, $|\varepsilon'(c_1)| = \wp'(c_1, c_2) \cdot |\varepsilon'(c_2)|$. This proves C4.

(b) Let $\mathbf{S}_1 = (\mathcal{C}_1, \sigma_1, \Rightarrow_1, \mathrm{me}_1, \wp_1)$, $\mathbf{S}_2 = (\mathcal{C}_2, \sigma_2, \Rightarrow_2, \mathrm{me}_2, \wp_2)$, and $\mathbf{S}_1 \bowtie \mathbf{S}_2 = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$. Let $\varepsilon_1 : \mathcal{C}_1 \rightarrow 2^{\mathcal{O}_1}$ and $\varepsilon_2 : \mathcal{C}_2 \rightarrow 2^{\mathcal{O}_2}$ be models of $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively. Let the mapping $\varepsilon : \mathcal{C} \rightarrow 2^{\mathcal{O}}$, where $\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2$ and $\mathcal{O} = \mathcal{O}_1 \times \mathcal{O}_2$, be defined as follows:

$$\varepsilon(c) \;=\; \varepsilon_1(c_1) \times \varepsilon_2(c_2), \quad \textbf{for all } c = (c_1, c_2) \in \mathcal{C}\,.$$

We now show that ε is a model of S. We first prove C1. Since $\varepsilon_1(c_1) \neq \emptyset$ for all classes $c_1 \in \mathcal{C}_1$ and $\varepsilon_2(c_2) \neq \emptyset$ for all classes $c_2 \in \mathcal{C}_2$, we get $\varepsilon(c) \neq \emptyset$ for all classes $c \in \mathcal{C}$. We next show C2 and C4. Let $c = (c_1, c_2), d = (d_1, d_2) \in \mathcal{C}$ with $c \Rightarrow d$. Without loss of generality, we can assume that $c_1 \Rightarrow_1 d_1$ and $c_2 = d_2$. Since ε_1 is a model of $\mathbf{S}_1$, it holds that $\varepsilon_1(c_1) \subseteq \varepsilon_1(d_1)$ and $|\varepsilon_1(c_1)| = \wp_1(c_1, d_1) \cdot |\varepsilon_1(d_1)|$. Hence, it immediately follows $\varepsilon(c) \subseteq \varepsilon(d)$ and $|\varepsilon(c)| = \wp(c, d) \cdot |\varepsilon(d)|$. We finally prove C3. Let $c, d \in \mathcal{C}$ be two distinct classes that belong to the same cluster $\mathcal{P} \in \bigcup \mathrm{me}(\mathcal{C})$. Without loss of generality, we can assume that $c_1, d_1 \in \mathcal{C}_1$ belong to the same cluster $\mathcal{P}_1 \in \bigcup \mathrm{me}_1(\mathcal{C}_1)$ and that $c_2 = d_2$. Since ε_1 is a model of $\mathbf{S}_1$, it holds that $\varepsilon_1(c_1) \cap \varepsilon_1(d_1) = \emptyset$. Thus, $\varepsilon(c) \cap \varepsilon(d) = \emptyset$. □ We next concentrate on the preservation of coherence. Recall that the coherence of a TPOB-instance $\mathbf{I} = (\pi, \nu)$ over a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$ depends on π, $\mathcal{C}$, me, and $\wp$. The operations of selection, restricted selection, intersection, union, and difference preserve coherence of instances, as they do not modify the input TPOB-schemas and they may only modify the input TPOB-instances by removing objects and changing value assignments to objects. Similarly, renaming and projection preserve coherence of instances, as they may only modify type and value assignments

to classes and objects, respectively. The result below shows that natural join, and thus also Cartesian product and conditional join, preserve coherence of instances. It also shows that the extraction operation preserves coherence of instances, when we do not remove any characteristic classes.

Theorem 13.0.5. *Let* $\mathbf{I}$*,* $\mathbf{I}_1$*, and* $\mathbf{I}_2$ *be TPOB-instances over the TPOB-schemas* $\mathbf{S}$*,* $\mathbf{S}_1$*, and* $\mathbf{S}_2$*, respectively, where* $\mathbf{I} = (\pi, \nu)$ *and* $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$*. Let* $\mathbf{C} \subseteq \mathcal{C}$ *such that* $\{c \in \mathcal{C} \mid c$ *is characteristic for* $\mathrm{ext}(d)(o)$ *for some* $d \in \mathbf{C}$ *and some* $o \in \pi(\mathbf{C})\} \subseteq \mathbf{C}$*. Let* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *be natural-join-compatible.*

(a) If $\mathbf{I}$ *is coherent, then* $\Xi_{\mathbf{C}}(\mathbf{I})$ *is coherent.*

(b) If $\mathbf{I}_1$ *and* $\mathbf{I}_2$ *are coherent, then* $\mathbf{I}_1 \bowtie \mathbf{I}_2$ *is coherent.*

Proof of Theorem 13.0.5. (a) Let $\mathbf{I} = (\pi, \nu)$ **and** $\Xi_{\mathbf{C}}(\mathbf{I}) = \mathbf{I}' = (\pi', \nu')$**. Let** $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$ **and** $\Xi_{\mathbf{C}}(\mathbf{S}) = \mathbf{S}' = (\mathcal{C}', \sigma', \Rightarrow', \mathrm{me}', \wp')$**. Towards a contradiction, suppose that** $\mathbf{I}'$ **is not coherent. That is, there exists a class** $c \in \mathcal{C}'$ **and an object** $o \in \pi'(c)$ **such that** $p_1, p_2 \in \mathrm{ext}'(c)(o)$ **with** $p_1 \neq p_2$**. Hence, there are at least two distinct classes** $d_1, d_2 \in \mathcal{C}'$ **such that (i)** $o \in (\pi')^\star(d_i)$ **and** $c\,(\Rightarrow')^\star\, d_i$**, and (ii)** d_i **is minimal under** $(\Rightarrow')^\star$ **with (i), and (iii)** p_i **is the product of edge probabilities from** c **up to** d_i**. As** $\mathbf{C}$ **contains all characteristic classes for** $\mathrm{ext}(c)(o)$**, there are at least two distinct classes** $d_i \in \mathcal{C}$ **such that (i)** $o \in \pi^\star(d_i)$ **and** $c \Rightarrow^\star d_i$**, and (ii)** d_i **is minimal under** $\Rightarrow^\star$ **with (i). This implies that** $p_1, p_2 \in \mathrm{ext}(c)(o)$**. But this contradicts** $\mathbf{S}$ **being coherent.**

(b) Let $\mathbf{S}_1 = (\mathcal{C}_1, \sigma_1, \Rightarrow_1, \mathrm{me}_1, \wp_1)$**,** $\mathbf{S}_2 = (\mathcal{C}_2, \sigma_2, \Rightarrow_2, \mathrm{me}_2, \wp_2)$**, and** $\mathbf{S}_1 \bowtie \mathbf{S}_2 = \mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$**. Let** $\mathbf{I}_1 = (\pi_1, \nu_1)$**,** $\mathbf{I}_2 = (\pi_2, \nu_2)$**, and** $\mathbf{I}_1 \bowtie \mathbf{I}_2 = \mathbf{I} = (\pi, \nu)$**. Towards a contradiction, suppose that** $\mathbf{I}$ **is not coherent. That is, there exists a class** $c = (c_1, c_2) \in \mathcal{C}$ **and an object** $o = (o_1, o_2) \in \pi(c)$ **such that** $p^1, p^2 \in \mathrm{ext}(c)(o)$ **with** $p^1 \neq p^2$**. Hence, there are classes** $d^1 = (d^1_1, d^1_2), d^2 = (d^2_1, d^2_2) \in \mathcal{C}$ **such that (i)** $o \in \pi^\star(d^i)$ **and** $c \Rightarrow^\star d^i$**, (ii)** d^i **is minimal under** $\Rightarrow^\star$ **with (i), and (iii)** p^i **is the product of edge probabilities from** c **up to** d^i**. Thus,** c**,** d^1**, and** d^2 **are pairwise distinct. Moreover,** $p^1 = p^1_1 \cdot p^1_2$ **and** $p^2 = p^2_1 \cdot p^2_2$**, where** p^i_j **is the product of edge probabilities from** c_j **up to** d^i_j**. Suppose now** $c_1 = d^1_1$**. Then,** $o \in \pi^\star((c_1, d^2_2))$**,** $c \Rightarrow^\star (c_1, d^2_2)$**, and** $(c_1, d^2_2) \Rightarrow^\star d^2$**. By the minimality of** d^2**, it then follows** $c_1 = d^1_1 = d^2_1$**, and thus** $p^1_1 = p^2_1 = 1$**. Moreover, if** $d^1_1 = d^2_1$**, then** $p^1_1 = p^2_1$**. Thus, as** $p_1 \neq p_2$**, we can assume without loss of generality that** c_1**,** d^1_1**, and** d^2_1 **are pairwise distinct. As** $\mathbf{S}_1$ **is coherent, there is some** $d^0_1 \in \mathcal{C}_1$ **such that** $o_1 \in \pi^\star(d^0_1)$**,** $c_1 \Rightarrow^\star d^0_1$**,** $d^0_1 \Rightarrow^\star d^1_1$**, and** $d^0_1 \Rightarrow^\star d^2_1$**. Without loss of generality, we can assume that**

$d_1^0 \neq d_1^1$**. Hence,** $o \in \pi^\star((d_1^0, d_2^1))$**,** $c \Rightarrow^\star (d_1^0, d_2^1)$**,** $(d_1^0, d_2^1) \Rightarrow^\star d^1$**, and** $(d_1^0, d_2^1) \neq d^1$**. But this contradicts** d^1 **being minimal under** $\Rightarrow^\star$ **with (i). Hence,** $\mathbf{I}$ **is coherent.** □

Chapter 14

Implicit TPOB-Instances

We now focus on how to efficiently implement the TPOB-instances introduced in Chapter 10 and the algebraic operations on TPOB-instances presented in Chapters 11 and 12.

The main problem with TPOB-instances is that the size of a TPOB-instance can be very large. Table 10.4 shows that a probability must be associated with each time point involved. But to state that a given package will arrive in St. Louis sometime between 5:30 pm and 6:30 pm may (if we reason at a minute by minute level) requires 60 time points to be specified (Table 10.4 only shows a couple of time points). The number of time points increase even more if they are defined in a calendar based on seconds instead of minutes. The second problem is that the large size of the TPOB-instances causes the cost of the algebraic operations to be potentially high.

In this and the following Chapter, we alleviate these problems by defining *implicit* TPOB-instances, together with algebraic operations on implicit TPOB-instances. An implicit TPOB-instance succinctly captures a potentially large TPOB value. We further show that these implicit operations correctly implement their counterparts in Chapters 11 and 12 by *solely considering the implicit (i.e. succinct) TPOB-instance rather than the explicit one*. Hence, all our implicit algebraic operators are usually much more efficient than their explicit counterparts. Furthermore, they immediately preserve consistency and coherence of schemas and instances, respectively.

The main idea behind implicit TPOB-instances is to use an implicit and thus more compact representation of the values of probabilistic types. We generalize a constraint-based approach due to Dekhtyar et al. [57].

For example, suppose we consider an explicit value of the form $v = \{(1, [0.001, 0.001]), \ldots, (1000, [0.001, 0.001)\}$. This says that v's value is one of $1, \ldots, 1000$ with some associated probability intervals. Instead of storing 1000 pairs of the form $(i, [0.001, 0.001])$, we could describe the range of values via the constraint $1 \leq t \leq 1000$. We can store the probability information in this case by merely saying that the uniform distribution is used to distribute probability values over the values $\{1, \ldots, 1000\}$ and that the entire probability mass 1 is distributed. Thus, our general idea is to use a constraint C to represent the time points at which some event may possibly hold, a lower and upper probability bound l, u, and a distribution δ. The technical definition of an *implicit value* will use these intuitions, but will include some additional twists.

We now first describe the concepts of constraints and of probability distribution functions. We then define implicit values of probabilistic types and implicit TPOB-instances.

14.1 Constraints

We use a constraint to implicitly specify a set of valid time points of a calendar, or a set of values of a classical type with a totally ordered domain, which is given by the set of all solutions to that constraint.

An *atomic constraint* for a calendar τ over the linear temporal hierarchy $T_1 \sqsupseteq \cdots \sqsupseteq T_n$ is either an *atomic time-value constraint* ($T_i \, \theta \, v_i$), where $\theta \in \{\leq, <, =, \neq, >, \geq\}$ and v_i is a time value of T_i, or an *atomic time-interval constraint* ($t_1 \sim t_2$), where $t_1, t_2 \in \mathrm{dom}(\tau)$ and $t_1 \leq_H t_2$. An *atomic constraint* for a classical type τ with a linear order $<$ on the domain $\mathrm{dom}(\tau)$ is either of the form $(\theta \, v)$, where $\theta \in \{\leq, <, =, \neq, >, \geq\}$ and $v \in \mathrm{dom}(\tau)$, or of the form $(v_1 \sim v_2)$, where $v_1, v_2 \in \mathrm{dom}(\tau)$ with $v_1 \leq v_2$. We use (t_1) (resp., (v_1)) to abbreviate $(t_1 \sim t_1)$ (resp., $(v_1 \sim v_1)$). A *constraint* for the above forms of classical types τ is a Boolean combination of atomic constraints for τ (that is, constructed from atomic constraints for τ by using the Boolean operators $\neg$ and $\wedge$).

We now define the semantics of a constraint, that is, the set of time points (resp., values) that it specifies. A time point $s = (s_1, \ldots, s_n) \in \mathrm{dom}(\tau)$ is a *solution* to an atomic constraint $(T_i \, \theta \, v_i)$ (resp., $(t_1 \sim t_2)$) for τ, denoted $s \models (T_i \, \theta \, v_i)$ (resp., $s \models (t_1 \sim t_2)$), iff $s_i \, \theta \, v_i$ (resp., $t_1 \leq_H s \leq_H t_2$). A value $s \in \mathrm{dom}(\tau)$ is a *solution* to an atomic constraint $(\theta \, v)$ (resp., $(v_1 \sim v_2)$) for τ, denoted $s \models (\theta \, v)$ (resp., $s \models (v_1 \sim v_2)$),

iff $s\,\theta\,v$ (resp., $v_1 \leq s \leq v_2$). The solutions $s \in \mathrm{dom}(\tau)$ to a constraint C for τ are inductively defined by (i) $s \models \neg C_1$ iff it is not the case that $s \models C_1$, and (ii) $s \models (C_1 \wedge C_2)$ iff $s \models C_1$ and $s \models C_2$, for all constraints C_1, C_2. We use $\mathrm{sol}(C)$ to denote the set of all solutions $s \in \mathrm{dom}(\tau)$ to the constraint C for τ.

14.2 Probability Distribution Functions

We now recall the well-known concept of a probability distribution function. In the sequel, we assume that S is a nonempty set of pairwise mutually exclusive events $s_1, \ldots, s_n$. A *probability distribution function* (or *distribution function*) over S is a mapping $\rho\colon S \to [0,1]$ such that $\sum_{s \in S} \rho(s) \leq 1$. The most widely used distribution function is the *uniform distribution* over S, denoted U_S, which is the function $U_S\colon S \to [0,1]$ defined by $U(s) = 1\,/\,n$ for all $s \in S$. The following are some other standard distribution functions (where we additionally assume that S has the linear order $s_1 < \ldots < s_n$):

- **The *geometric distribution* over S for $0 < p < 1$, denoted $G_{S,p}$, is the function $G_{S,p}\colon S \to [0,1]$ defined by $G_{S,p}(s_i) = p \cdot (1-p)^i$ for all $s_i \in S$.**
- **The *binomial distribution* over S for $0 < p < 1$, denoted $B_{S,p}$, is the function $B_{S,p}\colon S \to [0,1]$ defined by $B_{S,p}(s_i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$ for all $s_i \in S$.**
- **The *Poisson distribution* over S for $\lambda > 0$, denoted $P_{S,\lambda}$, is the function $P_{S,\lambda}\colon S \to [0,1]$ defined by $P_{S,\lambda}(s_i) = e^{-\lambda} \cdot \lambda^i\,/\,i\,!$ for all $s_i \in S$.**

14.3 Implicit Values of Probabilistic Types

An implicit value of a probabilistic type is a finite set of implicit tuples, a concept borrowed from [57]. An example of an implicit tuple (which we define formally below) is $(1 \leq t \leq 40, 1 \leq t \leq 100, 0.5, 1, u)$. This implicit tuple says that the valid time points are solutions of the first constraint $1 \leq t \leq 40$ and that a probability mass of 0.5 to 1 is uniformly distributed over the solutions of the second constraint $1 \leq t \leq 100$. From this, we can infer that there is a probability mass of 0.2 to 0.4 over the set of valid time points $\{1, \ldots, 40\}$. Using this intuition, we can now define implicit tuples formally.

Let τ be either a calendar or a classical type with a totally ordered domain. An *implicit tuple* for τ is a 5-tuple (C, D, l, u, δ), where C, D are constraints for τ with

$\emptyset \subset \mathrm{sol}(C) \subseteq \mathrm{sol}(D)$, l, u are reals with $0 \leq l \leq u \leq 1$, and δ is a distribution function over $\mathrm{sol}(D)$. If $\mathrm{sol}(C) = \mathrm{sol}(D)$, then we use $(\#)$ to abbreviate C.

The reader may wonder why both constraints C, D are needed. The reason is that C describes the valid values, while D describes the values from which the probabilities for solutions of C are *derived*. When selection operations are performed, we can merely restrict C to the selection condition (by performing an AND) rather than computing a new distribution (which would be needed if D were not used). Thus, the use of D eliminates the potentially expensive operation of computing new distributions every time a query is asked.

We now define *implicit values* of probabilistic types by induction as follows:

- **An implicit value of an atomic probabilistic type $[\![\tau]\!]$ is a finite set of implicit tuples for τ.**
- **An implicit value of a probabilistic type $[A_1 \colon \tau_1, \ldots, A_k \colon \tau_k]$ is of the form $[A_1 \colon v_1, \ldots, A_k \colon v_k]$, where $v_1, \ldots, v_k$ are either values or implicit values of $\tau_1, \ldots, \tau_k$.**

In the sequel, the values of probabilistic types introduced in Section 9.3 are called *explicit values*, to better distinguish them from the above concept of implicit values of probabilistic types. An implicit value of a probabilistic type in the Package Example is given below.

Example 14.3.1. An implicit value of the atomic probabilistic type $[\![\mathsf{time}]\!]$, where time is the calendar over $hour \sqsupseteq minute$, is given by $v = ((\#), ((12,00) \sim (12,59)), 0.5, 1, U)$. □

Observe now that every implicit value v of an atomic probabilistic type $[\![\tau]\!]$ has a unique equivalent explicit value $\varepsilon(v)$, which is defined by $\varepsilon(v) = \{(v', [l \cdot \delta(v'), u \cdot \delta(v')]) \mid \exists (C, D, l, u, \delta) \in v \colon v' \in \mathrm{sol}(C)\}$. ε is a *implicit to explicit transformation*. Conversely, every explicit value of $[\![\tau]\!]$ has at least one equivalent implicit value. The following example illustrates the relationship between implicit values v and their equivalent explicit values $\varepsilon(v)$.

Example 14.3.2. The implicit value v given in Example 14.3.1 has the equivalent explicit value $\varepsilon(v) = \{((12,00), [\frac{0.5}{60}, \frac{1}{60}]), ((12,01), [\frac{1}{60}]), ((12,02), [\frac{1}{60}]), \ldots, ((12,59), [\frac{1}{60}])\}$. *Note that instead of explicitly expressing a probability interval for each of 60 time points, the implicit value shown in Example 14.3.1 captures this explicit information via single statement.* Hence, implicit values are far more succinct than explicit ones. □

As every implicit value has its explicit counterpart, we can easily extend the notion of consistency to implicit values as follows. An implicit value $v = \{(C_1, D_1, l_1, u_1, \delta_1), \ldots, (C_n, D_n, l_n, u_n, \delta_n)\}$ **of an atomic probabilistic type is *consistent* iff** $\mathrm{sol}(C_i \wedge C_j) = \emptyset$ **for all** $i, j \in \{1, \ldots, n\}$ **with** $i \neq j$**, and it holds that** $\sum_{i=1}^{n} \sum_{v_i \in \mathrm{sol}(C_i)} l_i \cdot \delta_i(v_i) \leq 1 \leq \sum_{i=1}^{n} \sum_{v_i \in \mathrm{sol}(C_i)} u_i \cdot \delta_i(v_i)$**. An implicit value** v **of a probabilistic type is *consistent* iff all contained implicit values of atomic probabilistic types are consistent.**

14.4 Implicit TPOB-Instances

Implicit TPOB-instances associate an implicit value of appropriate probabilistic type with each object (rather than an explicit one). Formally, an *implicit TPOB-instance* over a consistent TPOB-schema $\mathbf{S}=(\mathcal{C}, \sigma, \mathrm{me}, \wp)$ **is a pair** $\mathbf{I} = (\pi, \nu)$**, where (i)** $\pi : \mathcal{C} \to 2^{\mathcal{O}}$ **maps each class** c **to a finite subset of** $\mathcal{O}$ **such that** $\pi(c_1) \cap \pi(c_2) = \emptyset$ **for all distinct** $c_1, c_2 \in \mathcal{C}$**, and (ii)** ν **maps each oid** $o \in \pi(c)$**,** $c \in \mathcal{C}$**, to an implicit value of type** $\sigma^\star(c)$**. We then define** $\pi(\mathcal{C})$**, the mapping** $\pi^\star$**, the concept of a probabilistic extent of a class, and the notions of coherence and consistency for implicit TPOB-instances as in Section 10.3. In the sequel, we write *explicit TPOB-instance* to refer to the TPOB-instances defined in Section 10.3.**

Every implicit TPOB-instance $\mathbf{I}$ **has a unique equivalent explicit TPOB-instance** $\varepsilon(\mathbf{I})$**, which is obtained from** $\mathbf{I}$ **by replacing each contained implicit value** v **of an atomic probabilistic type by its explicit counterpart** $\varepsilon(v)$**. Conversely, every explicit TPOB-instance has at least one equivalent implicit TPOB-instance.**

Example 14.4.1. An implicit TPOB-instance $\mathbf{I} = (\pi, \nu)$ over the TPOB-schema $\mathbf{S}$ of Example 10.1.1 is given by the mapping π in Table 10.3 and the value assignment ν in Table 14.1. □

Table 14.1: Value assignment ν

o	$\nu(o)$
o_3	[Origin: Rome, Destination: Boston, Delivery: $\{((\#), ((18,00) \sim (18,31)), .4, .9, U)$, $((\#), ((12,00) \sim (12,31)), .2, .3, G)\}$, Time: $\{((\#), ((8,00) \sim (8,10)), .4, 1, G)\}$]
o_5	[Origin: Paris, Destination: San_Jose, Delivery: $\{((\#), ((12,00) \sim (12,15)),$ $.15, 1, U)\}$, Height: 60, Width: 50, Depth: 40, Contents: {photos, books}, Time: $\{(((12,00) \sim (12,05)), .3, 1, G)\}$, STOPone: [City: New_York, Arrive: $\{(((14,00) \sim (14,05)) \vee ((14,25) \sim (14,30)), ((14,00) \sim (14,30)), .3, 1, U)\}$, Shipment: $\{((\#), ((16,00) \sim (17,00)), .6, 1, U)\}$], STOPtwo: [City: ST_Louis, Arrive: $\{((\#), ((17,30) \sim (17,45)), .5, 1, U)\}$, Shipment: $\{((\#), ((18,00) \sim (18,30)), .6, 1, U)\}$]]

Chapter 15

The Implicit Algebra

In this Chapter, we define the algebraic operations of selection, restricted selection, natural join, and the set operations of intersection and union on implicit TPOB-instances. The operations of projection, extraction, Cartesian product, and conditional join are independent of the kind of values of each object, and thus defined in exactly the same way for explicit and implicit TPOB-instances. We show that each and every operation of the implicit TPOB-algebra correctly implements its explicit counterpart. Figure 1.7 provides a diagrammatic representation of what these results look like for unary operators (a similar figure can be shown for binary operators). For unary algebraic operators op, the correctness theorems are of the form $\varepsilon(op^i(\mathbf{I})) = op^e(\varepsilon(\mathbf{I}))$. We use op^i (resp. op^e) to denote the implicit (resp. explicit) versions of the operator op. Similar correctness results are also shown to hold for binary algebraic operators. Whenever we apply an algebraic operation op, we can tell by the type of argument (explicit TPOB instance or implicit) whether the operator is one from the explicit or from the implicit algebra. Hence, we will usually just write op instead of op^e, op^i as this can be determined from context.

15.1 Selection

In order to define the selection operation on implicit TPOB-instances, it is sufficient to define how to evaluate path expressions and how to assess the probability that an implicit value satisfies an atomic selection condition. The valuation of selection conditions, the satisfaction of probabilistic selection conditions, and the selection on implicit

TPOB-instances are then defined in the same way as in Section 11.1.2.

We first define how to evaluate path expressions. Let P be a path expression for the probabilistic type τ. The *valuation* of P under an implicit value v of τ, denoted $v.P$, is defined by:

- **if** $v = [A_1 \colon v_1, \ldots, A_k \colon v_k]$ **and** $P = A_i$**, then** $v.P = v_i$ **;**
- **if** $v = [A_1 \colon v_1, \ldots, A_k \colon v_k]$ **and** $P = A_i.R$**, then** $v.P = v_i.R$ **;**
- **If** $v = \{(C_1, D_1, l_1, u_1, \delta_1), \ldots, (C_k, D_k, l_k, u_k, \delta_k)\}$ **and** $P = [\![R]\!]$**, then** $v.P = \{(C_1, D_1, l_1, u_1, \delta_1, R), \ldots, (C_k, D_k, l_k, u_k, \delta_k, R)\}$**. We call such sets *generalized implicit values* of τ ;**
- **Otherwise,** $v.P$ **is undefined.**

Given an implicit TPOB-instance $\mathbf{I} = (\pi, \nu)$ over a TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$, we now define the meaning of an atomic selection condition ϕ w.r.t. an object o in I. As in Section 11.1.2, we associate with each ϕ a closed subinterval of $[0, 1]$, which describes the range for the probability that the object o in I satisfies ϕ. This probability is computed in a similar way as in Section 11.1.2, based on the translation ε from implicit to explicit values. More formally, the *probabilistic valuation* of ϕ w.r.t. I and $o \in \pi(\mathcal{C})$, denoted $\mathrm{prob}_{\mathbf{I},o}$, is defined as follows (where $\oplus$ is the disjunction strategy for mutual exclusion):

- $\mathrm{prob}_{\mathbf{I},o}(in(c)) = [\min(\mathrm{ext(c)}(o)), \max(\mathrm{ext(c)}(o))]$**.**
- **Let P be a path expression for the type of o. If $\nu(o).P$ is a value of a classical type, then define $V = \{((\#), (\nu(o)), 1, 1, U, P)\}$, else if $\nu(o).P$ is a generalized implicit value of an atomic probabilistic type, then define $V = \nu(o).P$. Otherwise, V is undefined.**

$$\mathrm{prob}_{\mathbf{I},o}(P\,\theta\,v) = \begin{cases} \bigoplus_{i=1}^{k} I_i & \textbf{if } V \textbf{ is defined} \\ \textbf{undefined} & \textbf{otherwise,} \end{cases}$$

where $I_1, \ldots, I_k$ are the intervals $[l \cdot \delta(w), u \cdot \delta(w)]$ such that $(C, D, l, u, \delta, S) \in V$, $w \in \mathrm{sol}(C)$, and $w.S\,\theta\,v$, if V is defined. Note that $\mathrm{prob}_{\mathbf{I},o}(P\,\theta\,v)$ is undefined, if some $w.S\,\theta\,v$ is undefined.

- **For each $i \in \{1, 2\}$, let P_i be a path expression for the type of o. If $\nu(o).P_i$ is a value of a classical type, then define $V_i = \{((\#), (\nu(o)), 1, 1, U, P_i)\}$, else if**

$\nu(o).P_i$ **is a generalized implicit value of an atomic probabilistic type, then define** $V_i = \nu(o).P_i$**. Otherwise,** V_i **is undefined. Then,**

$$\mathrm{prob}_{\mathbf{I},o}(P_1\,\theta_{\otimes}\,P_2) = \begin{cases} \bigoplus_{i=1}^{k} I_i & \textbf{if } V_1 \textbf{ and } V_2 \textbf{ are defined} \\ \textbf{undefined} & \textbf{otherwise,} \end{cases}$$

where $I_1, \ldots, I_k$ **is the list of all intervals** $[l_1 \cdot \delta_1(v_1), u_1 \cdot \delta_1(v_1)] \otimes [l_2 \cdot \delta_2(v_2), u_2 \cdot \delta_2(v_2)]$ **such that** $(C_i, D_i, l_i, u_i, \delta_i, S_i) \in V_i$**,** $v_i \in \mathrm{sol}(C_i)$**, and** $v_1.S_1\,\theta\,v_2.S_2$**, if** V_1 **and** V_2 **are defined. Observe that** $\mathrm{prob}_{\mathbf{I},o}(P_1\ \theta_{\otimes}\ P_2)$ **is undefined, if some** $v_1.S_1\,\theta\,v_2.S_2$ **is undefined.**

The following example shows some probabilistic valuations of atomic selection conditions.

Example 15.1.1. Consider the implicit TPOB-instance $\mathbf{I} = (\pi, \nu)$ of Example 14.4.1. The atomic selection condition $\phi = \mathsf{Delivery} > (18, 25)$ is assigned $[0, 0]$ and $[.075, .169]$ under $\mathrm{prob}_{\mathbf{I},o_5}$ and $\mathrm{prob}_{\mathbf{I},o_3}$, respectively. Here, $[.075, .169] = \bigoplus_{i=1}^{6}[.0125, .0281]$, where the six intervals $[.0125, .0281]$ are associated with the six time points $(18, 26), \ldots, (18, 31)$ of the value of object o_3 at the attribute Delivery. □

The following result shows that selection on implicit TPOB-instances correctly implements its counterpart on explicit TPOB-instances. That is, the mapping ε **commutes with** σ_ϕ**.**

Theorem 15.1.2 (correctness of selection)**.** *Let* $\mathbf{I}$ *be an implicit TPOB-instance over a TPOB-schema* $\mathbf{S}$*, and let* ϕ *be a probabilistic selection condition. Then,*

$$\sigma_\phi(\varepsilon(\mathbf{I})) \quad = \quad \varepsilon(\sigma_\phi(\mathbf{I}))\,. \tag{15.1}$$

or the proof of Theorem 15.1.2, we need the following lemma, which says that the valuation of path expressions under implicit values correctly implements the valuation of path expressions under explicit values. Here, the mapping ε **is extended to generalized implicit values as follows: Every generalized implicit value** $w = \{(C_1, D_1, l_1, u_1, \delta_1, S), \ldots, (C$ $D_k, l_k, u_k, \delta_k, S)\}$ **is associated with the generalized explicit value** $\varepsilon(w) = \{(w', I, S) \mid (w', I) \in \varepsilon(\{(C_1, D_1, l_1, u_1, \delta_1), \ldots, (C_k, D_k, l_k, u_k, \delta_k)\})\}$**.**

Lemma 15.1.3. *Let P be a path expression for the probabilistic type τ, and v be an implicit value of τ. Then,*

$$\varepsilon(v.P) \;=\; \varepsilon(v).P\,. \tag{15.2}$$

Proof. It is sufficient to show that $\varepsilon(v.P) = \varepsilon(v).P$ holds for every implicit value v of an atomic probabilistic type τ. Let $v = \{(C_1, D_1, l_1, u_1, \delta_1), \ldots, (C_k, D_k, l_k, u_k, \delta_k)\}$ and $P = [\![R]\!]$. Then,

$$\begin{aligned}
\varepsilon(v.P) &= \varepsilon(\{(C_1, D_1, l_1, u_1, \delta_1, R), \ldots, (C_k, D_k, l_k, u_k, \delta_k, R)\}) \\
&= \{(w', I, R) \mid (w', I) \in \varepsilon(\{(C_1, D_1, l_1, u_1, \delta_1), \ldots, (C_k, D_k, l_k, u_k, \delta_k)\})\} \\
&= \{(w', I, R) \mid (w', I) \in \varepsilon(v)\} \\
&= \varepsilon(v).P\,. \;\square
\end{aligned}$$

Proof of Theorem 15.1.2. It is sufficient to show that the valuation of atomic selection conditions with respect to implicit TPOB-instances is correct. Let $\mathbf{I} = (\pi, \nu)$ be an implicit TPOB-instance over the TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$, and let $o \in \pi(\mathcal{C})$. Let $\oplus$ be the disjunction strategy for mutual exclusion. Then,

- $\mathrm{prob}_{\mathbf{I},o}(in(c)) = [\min(\mathrm{ext(c)}(o)), \max(\mathrm{ext(c)}(o))] = \mathrm{prob}_{\varepsilon(\mathbf{I}),o}(in(c))$**.**
- **Let P be a path expression for the type of o. If $\nu(o).P$ is a value of a classical type, then define $V = \{((\#), (\nu(o)), 1, 1, U, P)\}$, else if $\nu(o).P$ is a generalized implicit value of an atomic probabilistic type, then define $V = \nu(o).P$. Otherwise, V is undefined.**

$$\mathrm{prob}_{\mathbf{I},o}(P\,\theta\,v) = \begin{cases} \bigoplus_{i=1}^{k} I_i & \textbf{if } V \textbf{ is defined} \\ \textbf{undefined} & \textbf{otherwise,} \end{cases}$$

where $I_1, \ldots, I_k$ are the intervals $[l \cdot \delta(w), u \cdot \delta(w)]$ such that $(C, D, l, u, \delta, S) \in V$, $w \in \mathrm{sol}(C)$, and $w.S\,\theta\,v$, if V is defined. That is, the intervals $[l', u']$ such that $(w, [l', u'], S) \in \varepsilon(V)$ and $w.S\,\theta\,v$, if V is defined. By Lemma 15.1.3 and as $\varepsilon(\{((\#), (\nu(o)), 1, 1, U, P)\}) = \{(\nu(o), [1, 1], P)\}$, it follows that

$$\mathrm{prob}_{\mathbf{I},o}(P\,\theta\,v) = \mathrm{prob}_{\varepsilon(\mathbf{I}),o}(P\,\theta\,v)\,.$$

- **For each $i \in \{1,2\}$, let P_i be a path expression for the type of o. If $\nu(o).P_i$ is a value of a classical type, then define $V_i = \{((\#),(\nu(o)),1,1,U,P_i)\}$, else if $\nu(o).P_i$ is a generalized implicit value of an atomic probabilistic type, then define $V_i = \nu(o).P_i$. Otherwise, V_i is undefined. Then,**

$$\mathrm{prob}_{\mathbf{I},o}(P_1\,\theta_{\otimes}\,P_2) = \begin{cases} \bigoplus_{i=1}^{k} I_i & \text{if } V_1 \text{ and } V_2 \text{ are defined} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

**where $I_1,\ldots,I_k$ is the list of all intervals $[l_1\cdot\delta_1(v_1), u_1\cdot\delta_1(v_1)]\otimes[l_2\cdot\delta_2(v_2), u_2\cdot\delta_2(v_2)]$ such that $(C_i,D_i,l_i,u_i,\delta_i,S_i)\in V_i$, $v_i\in\mathrm{sol}(C_i)$, and $v_1.S_1\,\theta\,v_2.S_2$, if V_1 and V_2 are defined. That is, the list of all intervals $[l_1',u_1']\otimes[l_2',u_2']$ such that $(v_i,[l_i',u_i'],S_i)\in\varepsilon(V_i)$ and $v_1.S_1\,\theta\,v_2.S_2$, if V_1 and V_2 are defined. By Lemma 15.1.3 and as $\varepsilon(\{((\#),$
$(\nu(o)),1,1,U,P_i)\}) = \{(\nu(o),[1,1],P_i)\}$, it follows that**

$$\mathrm{prob}_{\mathbf{I},o}(P_1\,\theta_{\otimes}\,P_2) = \mathrm{prob}_{\varepsilon(\mathbf{I}),o}(P_1\,\theta_{\otimes}\,P_2)\,.$$

This shows that $\mathrm{prob}_{\mathbf{I},o}(\phi) = \mathrm{prob}_{\varepsilon(\mathbf{I}),o}(\phi)$ for all atomic selection conditions ϕ. Notice that this statement also includes that $\mathrm{prob}_{\mathbf{I},o}(\phi)$ is defined iff $\mathrm{prob}_{\varepsilon(\mathbf{I}),o}(\phi)$ is defined. □

15.2 Restricted Selection

In order to define the restricted selection operation on implicit TPOB-instances, it suffices to define the restricted selection on implicit values. Restricted selection on implicit TPOB-instances is then defined in the same way as in Section 11.2.

Let τ be a probabilistic tuple type, and let $v = [A_1\colon v_1,\ldots,A_i\colon v_i,\ldots,A_k\colon v_k]$ be an implicit value of τ. Let ϕ be an atomic selection condition of the form $P\,\theta\,w$, where P is a path expression for τ. The *restricted selection* on v with respect to ϕ, denoted $\sigma^r_\phi(v)$, is defined by:

- **If v_i is a value of a classical type, $P = A_i$, and $v_i\,\theta\,w$, then $\sigma^r_\phi(v) = v$.**
- **If v_i is an implicit value of an atomic probabilistic type, and $P = A_i$, then $\sigma^r_\phi(v)$ is**

obtained from v by replacing v_i by $\{((\theta\, w)\ \wedge C, D, l, u, \delta) \mid (C, D, l, u, \delta)\ \in v_i,\ \mathrm{sol}\ ((\theta\, w)\ \wedge C) \neq \emptyset\}$.

- **If v_i is an implicit value of a probabilistic tuple type, and $P = A_i.R$, then $\sigma^r_\phi(v)$ is obtained from v by replacing v_i by $\sigma^r_{R\,\theta\, w}(v_i)$.**
- **Otherwise, $\sigma^r_\phi(v)$ is undefined.**

Example 15.2.1. Consider the implicit TPOB-instance $\mathbf{I} = (\pi, \nu)$ of Example 14.4.1 and the atomic selection condition $\phi = \mathsf{Delivery} > (18, 25)$. The restricted selection on $\mathbf{I}$ with respect to ϕ is given by the implicit TPOB-instance $\sigma^r_\phi(\mathbf{I}) = (\pi', \nu')$, where $\pi'(\mathsf{Priority}) = \{o_3\}$, $\pi'(c) = \emptyset$ for all other classes c, and $\nu'(o_3) = [\mathsf{Origin}\colon\ \mathsf{Rome}, \mathsf{Destination}\colon \mathsf{Boston}, \mathsf{Delivery}$ $\colon \{(((18, 26) \sim (18, 31)), ((18, 00) \sim (18, 31)), .4, .9, U)\}$, $\mathsf{Time}\ \colon \{((\#), ((8, 00)\ \sim (8, 10))$ $, .4, 1, G)\}]$. □

The following theorem shows that restricted selection on implicit instances correctly implements its counterpart on explicit instances. That is, the mapping ε commutes with σ^r_ϕ.

Theorem 15.2.2 (correctness of restricted selection)**.** *Let* $\mathbf{I}$ *be an implicit TPOB-instance over a TPOB-schema* $\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$*. Let* ϕ *be an atomic selection condition of the form* $P\,\theta\, v$*, where* P *is a path expression for every* $\sigma(c)$ *with* $c \in \mathcal{C}$*. Then,*

$$\sigma^r_\phi(\varepsilon(\mathbf{I})) \quad = \quad \varepsilon(\sigma^r_\phi(\mathbf{I}))\,. \tag{15.3}$$

Proof of Theorem 15.2.2. It is sufficient to show that the restricted selection on implicit values of atomic probabilistic types is correct. Let $\tau = [A_1\colon \tau_1, \ldots, A_k\colon \tau_k]$ be a probabilistic tuple type, and let $v = [A_1\colon v_1, \ldots, A_k\colon v_k]$ be an implicit value of τ. Let $\phi = A_i.C$, where $i \in \{1, \ldots, k\}$ and C is a constraint, and let τ_i be an atomic probabilistic type. Then,

$$\begin{aligned}
& \varepsilon(\{(C \wedge C', D, l, u, \delta) \mid (C', D, l, u, \delta) \in v_i,\ \mathrm{sol}(C \wedge C') \neq \emptyset\}) \\
= \ & \{(v_i', [l \cdot \delta(v_i'), u \cdot \delta(v_i')]) \mid \exists (C', D, l, u, \delta) \in v_i\colon\ v_i' \in \mathrm{sol}(C \wedge C')\} \\
= \ & \{(v_i', [l', u']) \in \varepsilon(v_i) \mid v_i' \in \mathrm{sol}(C)\}\,.
\end{aligned}$$

This shows that $\varepsilon(\sigma^r_\phi(v)) = \sigma^r_\phi(\varepsilon(v))$. □

15.3 Renaming

To define renaming on implicit TPOB-instances, we need to define renaming on implicit values, which is then extended to implicit TPOB-instances in the same way as in Section 11.3.

Let C be a constraint for the classical type τ, and let N be a renaming condition for τ. The *renaming* on C with respect to N, denoted $\delta_N(C)$, is obtained from C by replacing every value v_i in C by $\delta_N(v_i)$.

Let N be a renaming condition of the form $P \leftarrow P'$ for the probabilistic tuple type $\tau = [A_1 \colon \tau_1, \ldots, A_n \colon \tau_n]$. Let $v = [A_1 \colon v_1, \ldots, A_n \colon v_n]$ be an implicit value of τ. The *renaming* on v with respect to N, denoted $\delta_N(v)$, is defined by:

- **If $P = A_i$ and $P' = A_i'$, then $\delta_N(v)$ is obtained from v by replacing A_i by A_i'.**
- **If $P = A_i.[\![R]\!]$, $P' = A_i.[\![R']\!]$, and v_i is a value of an atomic probabilistic type, then $\delta_N(v)$ is obtained from v by replacing every $(C, D, l, u, \rho) \in v_i$ by $(\delta_{R \leftarrow R'}(C), \delta_{R \leftarrow R'}(D), l, u, \rho \circ \delta^{-1}_{R \leftarrow R'})$, where $\delta^{-1}_{R \leftarrow R'}$ denotes the inverse to $\delta_{R \leftarrow R'} : \mathrm{sol}(D) \rightarrow \mathrm{sol}(\delta_{R \leftarrow R'}(D))$.**
- **If $P = A_i.R$, $P' = A_i.R'$, and v_i is not a value of an atomic probabilistic type, then $\delta_N(v)$ is obtained from v by replacing v_i by $\delta_{R \leftarrow R'}(v_i)$.**

Let $N = P_1, \ldots, P_l \leftarrow P_1', \ldots, P_l'$ be a renaming condition for τ. The *renaming* on v with respect to N, denoted $\delta_N(v)$, is defined as the simultaneous renaming on v with respect to all $P_i \leftarrow P_i'$.

The following result shows that the renaming on implicit TPOB-instances correctly implements its counterpart on explicit TPOB-instances. That is, the mapping ε commutes with δ_N.

Theorem 15.3.1 (correctness of renaming)**.** *Let* $\mathbf{I}$ *be an implicit TPOB-instance over the TPOB-schema* $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$*, and let* N *be a renaming condition for every* $\sigma(c)$ *with* $c \in \mathcal{C}$*. Then,*

$$\delta_N(\varepsilon(\mathbf{I})) \;=\; \varepsilon(\delta_N(\mathbf{I}))\,.$$

Proof of Theorem 15.3.1. It is sufficient to show that the renaming of single attributes inside implicit values of atomic probabilistic types is correct. Let N be a renaming condition of the form $A_i.[\![R]\!] \leftarrow A_i.[\![R']\!]$ for the probabilistic tuple type $\tau =$

$[A_1 \colon \tau_1, \ldots, A_n \colon \tau_n]$**, where** τ_i **is an atomic probabilistic type, and let** $v = [A_1 \colon v_1, \ldots, A_n \colon v_n]$ **be an implicit value of** τ**. Then,**

$$\begin{aligned}
& \varepsilon(\{(\delta_{R \leftarrow R'}(C), \delta_{R \leftarrow R'}(D), l, u, \rho \circ \delta^{-1}_{R \leftarrow R'}) \mid (C, D, l, u, \rho) \in v_i\}) \\
= \; & \{(v_i{}'', [l \cdot \rho(\delta^{-1}_{R \leftarrow R'}(v_i{}'')), u \cdot \rho(\delta^{-1}_{R \leftarrow R'}(v_i{}''))]) \mid \exists (C, D, l, u, \rho) \in v_i \colon v_i{}'' \in \mathrm{sol} \\
& (\delta_{R \leftarrow R'}(C))\} \\
= \; & \{(\delta_{R \leftarrow R'}(v_i{}'), [l \cdot \rho(v_i{}'), u \cdot \rho(v_i{}')]) \mid \exists (C, D, l, u, \rho) \in v_i \colon v_i{}' \in \mathrm{sol}(C)\} \\
= \; & \{(\delta_{R \leftarrow R'}(v_i{}'), [l', u']) \mid (v_i{}', [l', u']) \in \varepsilon(v_i)\} .
\end{aligned}$$

This shows that $\varepsilon(\delta_N(v)) = \delta_N(\varepsilon(v))$**.** $\square$

15.4 Natural Join

In order to define the natural join operation on implicit TPOB-instances, we need to define the intersection of two implicit values. The join of two implicit values and the join of two TPOB-instances are then defined in the same way as in Section 12.1.

Let v_1 **and** v_2 **be either two values of the same classical type** τ **or two implicit values of the same probabilistic type** τ**, and let** $\otimes$ **be a conjunction strategy. The *intersection* of** v_1 **and** v_2 **under** $\otimes$**, denoted** $v_1 \cap_\otimes v_2$**, is inductively defined as follows:**

- **If** τ **is a classical type and** $v_1 = v_2$**, then** $v_1 \cap_\otimes v_2 = v_1$**.**
- **If** τ **is an atomic probabilistic type and** $w \neq \emptyset$**, then** $v_1 \cap_\otimes v_2 = w$**, where:**

$$\begin{aligned}
w \; = \; & \{((\#), (v), l, u, U) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 \colon \\
& \qquad v \in \mathrm{sol}(C_1 \wedge C_2),\ [l, u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \otimes [l_2 \cdot \delta_2(v), u_2 \\
& \cdot \delta_2(v)]\} .
\end{aligned}$$

- **If** τ **is a probabilistic tuple type over the set of top-level attributes A, and all** $v_1.A \cap_\otimes v_2.A$ **are defined, then** $(v_1 \cap_\otimes v_2).A = v_1.A \cap_\otimes v_2.A$ **for all** $A \in \mathbf{A}$**.**
- **Otherwise,** $v_1 \cap_\otimes v_2$ **is undefined.**

The following example illustrates the intersection of two implicit values of atomic probabilistic types.

Example 15.4.1. Let time be the standard calendar w.r.t. $hour \sqsupseteq minute$, and let $\otimes_{in}$ be the conjunction strategy for independence. Consider now the following implicit values of the atomic probabilistic type $[\![\mathsf{time}]\!]$:

$$v_1 = \{(((9,41) \sim (9,45)), ((9,41) \sim (9,50)), .5, 1, U)\},$$

$$v_2 = \{(\#), ((9,44) \sim (9,48)), .2, 1, U)\}$$

Then, $v_1 \cap_{\otimes_{in}} v_2 = \{((\#), ((9,44)), .04, .004, U), ((\#), ((9,45)), .04, .004, U)\}$. □

The result below shows that join on implicit TPOB-instances is correct, i.e. the mapping ε commutes with $\bowtie_{\otimes}$.

Theorem 15.4.2 (correctness of natural join)**.** *Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two implicit TPOB-instances over the natural-join-compatible TPOB-schemas $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively. Let $\otimes$ be a conjunction strategy. Then,*

$$\varepsilon(\mathbf{I}_1) \bowtie_{\otimes} \varepsilon(\mathbf{I}_2) \quad = \quad \varepsilon(\mathbf{I}_1 \bowtie_{\otimes} \mathbf{I}_2)\,. \tag{15.4}$$

Proof of Theorem 15.4.2. It is sufficient to show that the intersection of two implicit values of the same atomic probabilistic type is correct. Let v_1 and v_2 be two values of the same atomic probabilistic type, and let $\otimes$ be a conjunction strategy. Then,

$$\begin{aligned}
&\varepsilon(\{((\#), (v), l, u, U) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 : \\
&\qquad v \in \mathrm{sol}(C_1 \wedge C_2),\ [l,u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \otimes [l_2 \cdot \delta_2(v), u_2 \cdot \delta_2(v)]\}) \\
= \quad &\{(v, [l,u]) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 : \\
&\qquad v \in \mathrm{sol}(C_1 \wedge C_2),\ [l,u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \otimes [l_2 \cdot \delta_2(v), u_2 \cdot \delta_2(v)]\} \\
= \quad &\{(v, [l_1', u_1'] \otimes [l_2', u_2']) \mid (v, [l_1', u_1']) \in \varepsilon(v_1), (v, [l_2', u_2']) \in \varepsilon(v_2)\}\,.
\end{aligned}$$

This shows that $\varepsilon(v_1 \cap_{\otimes} v_2)$ is defined iff $\varepsilon(v_1) \cap_{\otimes} \varepsilon(v_2)$ is defined. Moreover, if they are both defined, then $\varepsilon(v_1 \cap_{\otimes} v_2) = \varepsilon(v_1) \cap_{\otimes} \varepsilon(v_2)$. □

15.5 Intersection, Union, and Difference

To define the intersection, union, and difference of two TPOB-instances, we need to define the intersection, union, and difference, respectively, of two implicit values, which are then extended to implicit TPOB-instances using methods similar to those of Section 12.3. Intersection of two implicit values was defined in the preceding section. Union and difference of two implicit values are defined below.

Let v_1 and v_2 be either two values of the same classical type τ or two implicit values of the same probabilistic type τ, and let $\oplus$ (resp., $\ominus$) be a disjunction (resp., difference) strategy. The *union* of v_1 and v_2 under $\oplus$, denoted $v_1 \cup_\oplus v_2$, is inductively defined as follows:

- **If τ is a classical type and $v_1 = v_2$, then $v_1 \cup_\oplus v_2 = v_1$.**
- **If τ is an atomic probabilistic type, then**

$$
\begin{aligned}
v_1 \cup_\oplus v_2 \;=\; & \{(C_1 \wedge \neg\widehat{C}_2, D_1, l_1, u_1, \delta_1) \mid (C_1, D_1, l_1, u_1, \delta_1) \in v_1,\ \mathrm{sol} \\
& (C_1 \wedge \neg\widehat{C}_2) \neq \emptyset\} \cup \\
& \{(C_2 \wedge \neg\widehat{C}_1, D_2, l_2, u_2, \delta_2) \mid (C_2, D_2, l_2, u_2, \delta_2) \in v_2,\ \mathrm{sol} \\
& (C_2 \wedge \neg\widehat{C}_1) \neq \emptyset\} \cup \\
& \{((\#), (v), l, u, U) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in \\
& v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 : \\
& \qquad v \in \mathrm{sol}(C_1 \wedge C_2),\ [l, u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \oplus \\
& [l_2 \cdot \delta_2(v), u_2 \cdot \delta_2(v)]\}\,,
\end{aligned}
$$

  where $\widehat{C}_i$, $i \in \{1, 2\}$, denotes the logical disjunction of all C_i such that $(C_i, D_i, l_i, u_i, \delta_i) \in v_i$.

- **If τ is a probabilistic tuple type over the set of top-level attributes A and all $v_1.A \cup_\oplus v_2.A$ are defined, then $(v_1 \cup_\oplus v_2).A = v_1.A \cup_\oplus v_2.A$ for all $A \in$ A.**
- **Otherwise, $v_1 \cup_\oplus v_2$ is undefined.**

The *difference* of v_1 and v_2 under $\ominus$, denoted $v_1 -_\ominus v_2$, is inductively defined as follows:

- **If τ is a classical type and $v_1 = v_2$, then $v_1 -_\ominus v_2 = v_1$.**

- **If τ is an atomic probabilistic type, then**

$$\begin{aligned} v_1 -_\ominus v_2 \;=\; & \{(C_1 \wedge \neg\widehat{C}_2, D_1, l_1, u_1, \delta_1) \mid (C_1, D_1, l_1, u_1, \delta_1) \in v_1,\ \mathrm{sol} \\ & (C_1 \wedge \neg\widehat{C}_2) \neq \emptyset\} \cup \\ & \{((\#), (v), l, u, U) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in \\ & v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 : \\ & \qquad v \in \mathrm{sol}(C_1 \wedge C_2),\ [l, u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \ominus \\ & [l_2 \cdot \delta_2(v), u_2 \cdot \delta_2(v)]\}\,, \end{aligned}$$

  where $\widehat{C}_i$, $i \in \{1, 2\}$, denotes the logical disjunction of all C_i such that $(C_i, D_i, l_i, u_i, \delta_i) \in v_i$.

- **If τ is a probabilistic tuple type over the set of top-level attributes $\mathbf{A}$ and all $v_1.A -_\ominus v_2.A$ are defined, then $(v_1 -_\ominus v_2).A = v_1.A -_\ominus v_2.A$ for all $A \in \mathbf{A}$.**

- **Otherwise, $v_1 -_\ominus v_2$ is undefined.**

The following theorem shows that the intersection, union, and difference of implicit TPOB-instances correctly implement their counterparts on explicit TPOB-instances. That is, the mapping ε commutes with $\cap_\otimes$, $\cup_\oplus$, and $-_\ominus$, respectively.

Theorem 15.5.1 (correctness of intersection, union, and difference)**.** *Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two implicit TPOB-instances over the same TPOB-schema $\mathbf{S}$, and let $\otimes$ (resp., $\oplus$, $\ominus$) be a conjunction (resp., disjunction, difference) strategy. Then,*

$$\varepsilon(\mathbf{I}_1) \cap_\otimes \varepsilon(\mathbf{I}_2) \;=\; \varepsilon(\mathbf{I}_1 \cap_\otimes \mathbf{I}_2)\,, \tag{15.5}$$

$$\varepsilon(\mathbf{I}_1) \cup_\oplus \varepsilon(\mathbf{I}_2) \;=\; \varepsilon(\mathbf{I}_1 \cup_\oplus \mathbf{I}_2)\,, \tag{15.6}$$

$$\varepsilon(\mathbf{I}_1) -_\ominus \varepsilon(\mathbf{I}_2) \;=\; \varepsilon(\mathbf{I}_1 -_\ominus \mathbf{I}_2)\,. \tag{15.7}$$

Proof of Theorem 15.5.1. Equation (15.5) follows immediately from the proof of Theorem 15.4.2. We next prove Equation (15.6). It is sufficient to show that the union of two implicit values of the same atomic probabilistic type is correct. Let v_1 and v_2 be two values of the same atomic probabilistic type, and let $\oplus$ be a disjunction strategy.

Then,

$$
\begin{aligned}
& \varepsilon(v_1 \cup_{\oplus} v_2) \\
= \;& \varepsilon(\{(C_1 \wedge \neg\widehat{C}_2, D_1, l_1, u_1, \delta_1) \mid (C_1, D_1, l_1, u_1, \delta_1) \in v_1,\, \mathrm{sol}(C_1 \wedge \neg\widehat{C}_2) \neq \emptyset\}) \cup \\
& \varepsilon(\{(C_2 \wedge \neg\widehat{C}_1, D_2, l_2, u_2, \delta_2) \mid (C_2, D_2, l_2, u_2, \delta_2) \in v_2,\, \mathrm{sol}(C_2 \wedge \neg\widehat{C}_1) \neq \emptyset\}) \cup \\
& \varepsilon(\{((\#), (v), l, u, U) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 : \\
& \qquad v \in \mathrm{sol}(C_1 \wedge C_2),\, [l, u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \oplus [l_2 \cdot \delta_2(v), u_2 \cdot \delta_2(v)]\}) \\
= \;& \{(v_1', [l_1 \cdot \delta_1(v_1'), u_1 \cdot \delta_1(v_1')]) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in v_1 :\, v_1' \in \mathrm{sol}(C_1) - \mathrm{sol}(\widehat{C}_2)\} \cup \\
& \{(v_2', [l_2 \cdot \delta_2(v_2'), u_2 \cdot \delta_2(v_2')]) \mid \exists (C_2, D_2, l_2, u_2, \delta_2) \in v_2 :\, v_2' \in \mathrm{sol}(C_2) - \mathrm{sol}(\widehat{C}_1)\} \cup \\
& \{(v, [l_1', u_1'] \oplus [l_2', u_2']) \mid (v, [l_1', u_1']) \in \varepsilon(v_1), (v, [l_2', u_2']) \in \varepsilon(v_2)\} \\
= \;& \{(v_1', [l_1', u_1']) \in \varepsilon(v_1) \mid v_1' \notin \mathrm{sol}(\widehat{C}_2)\} \cup \{(v_2', [l_2', u_2']) \in \varepsilon(v_2) \mid v_2' \notin \mathrm{sol}(\widehat{C}_1)\} \cup \\
& \{(v, [l_1', u_1'] \oplus [l_2', u_2']) \mid (v, [l_1', u_1']) \in \varepsilon(v_1), (v, [l_2', u_2']) \in \varepsilon(v_2)\} \\
= \;& \varepsilon(v_1) \cup_{\oplus} \varepsilon(v_2)\,.
\end{aligned}
$$

We finally prove Equation (15.7). Again, it is sufficient to show that the difference of two implicit values of the same atomic probabilistic type is correct. Let v_1 and v_2 be two values of the same atomic probabilistic type, and let $\ominus$ be a difference strategy. Then,

$$
\begin{aligned}
& \varepsilon(v_1 -_{\ominus} v_2) \\
= \;& \varepsilon(\{(C_1 \wedge \neg\widehat{C}_2, D_1, l_1, u_1, \delta_1) \mid (C_1, D_1, l_1, u_1, \delta_1) \in v_1,\, \mathrm{sol}(C_1 \wedge \neg\widehat{C}_2) \neq \emptyset\}) \cup \\
& \varepsilon(\{((\#), (v), l, u, U) \mid \exists (C_1, D_1, l_1, u_1, \delta_1) \in v_1, (C_2, D_2, l_2, u_2, \delta_2) \in v_2 : \\
& \qquad v \in \mathrm{sol}(C_1 \wedge C_2),\, [l, u] = [l_1 \cdot \delta_1(v), u_1 \cdot \delta_1(v)] \ominus [l_2 \cdot \delta_2(v), u_2 \cdot \delta_2(v)]\}) \\
= \;& \{(v_1', [l_1', u_1']) \in \varepsilon(v_1) \mid v_1' \notin \mathrm{sol}(\widehat{C}_2)\} \cup \\
& \{(v, [l_1', u_1'] \ominus [l_2', u_2']) \mid (v, [l_1', u_1']) \in \varepsilon(v_1), (v, [l_2', u_2']) \in \varepsilon(v_2)\} \\
= \;& \varepsilon(v_1) -_{\ominus} \varepsilon(v_2)\,. \;\square
\end{aligned}
$$

15.6 Compression Functions

The implicit operations of natural join, intersection, union, and difference may generate implicit TPOB-instances that contain a large number of implicit tuples. Adopting

an idea from [57], we now define compression functions through which such implicit TPOB-instances can be made more compact.

A *compression function* Γ for an atomic probabilistic type τ is a function that maps every implicit value v of τ to an implicit value $\Gamma(v)$ of τ such that (i) $|\Gamma(v)| \leq |v|$, and (ii) there exists a bijection between $\varepsilon(v)$ and $\varepsilon(\Gamma(v))$ that maps each $(v, [l, u]) \in \varepsilon(v)$ to a pair $(v, [l, u']) \in \varepsilon(\Gamma(v))$ such that $l \leq u' \leq u$.

Example 15.6.1. Let τ be an atomic probabilistic type. The *same-distribution compression function* Γ maps every implicit value v of τ to the implicit value $\Gamma(v)$, which is obtained from v by iteratively replacing any two distinct $(C_1, D_1, l, u, \delta), (C_2, D_2, l, u, \delta) \in v$ with $\mathrm{sol}(D_1) = \mathrm{sol}(D_2)$ by $(C_1 \vee C_2, D_1, l, u, \delta)$.

We now define the compression of implicit values of probabilistic types. Here, we assume that for every atomic probabilistic type τ, we have some compression function Γ_τ. More formally, let v be either a value of a classical type τ, or an implicit value of a probabilistic type τ. The *compression* of v, denoted $\Gamma(v)$, is inductively defined as follows:

- **If τ is a classical type, then $\Gamma(v) = v$.**
- **If τ is an atomic probabilistic type, then $\Gamma(v) = \Gamma_\tau(v)$.**
- **If τ is a probabilistic tuple type over the set of top-level attributes $\mathbf{A}$, then $\Gamma(v).A = \Gamma(v.A)$ for all $A \in \mathbf{A}$.**

We finally define the compression of implicit TPOB-instances as follows. Let $\mathbf{I} = (\pi, \nu)$ be a TPOB-instance over the TPOB-schema $\mathbf{S} = (\mathcal{C}, \sigma, \Rightarrow, \mathrm{me}, \wp)$. The *compression* of $\mathbf{I}$, denoted $\Gamma(\mathbf{I})$, is defined as the TPOB-instance (π, ν') over $\mathbf{S}$, where $\nu'(o) = \Gamma(\nu(o))$ for all $o \in \pi(\mathcal{C})$.

Chapter 16

Conclusions

There are numerous applications where object models are naturally used and where temporal uncertainty is a critical part of the application. A natural place to start is shipping and transportation companies. These companies utilize numerous different kinds of resources (airplanes, ships, and trucks, all with different capacities, fuel requirements, fuel consumption, servicing needs, and other properties) and ship hundreds of thousands of items daily. The properties of these vehicles vary dramatically (as the properties of ships, planes, and trucks are all very different). The items being shipped also have diverse properties (commercial shippers such as CSX ship everything from packages of papers to machine parts, vehicles, and hazardous materials, each with very different properties). Such shippers use prediction programs that provide temporal projections that are characterized by uncertainty. These projections (and the other data described above) are useful in a wide range of decision making activities ranging from scheduling to asset allocation problems. The ability to query the above data in the presence of such temporally uncertain projections is critical for such decision making.

In this thesis, we have made a first attempt to deal with temporal uncertainty in object-based systems. We have provided a data model and algebraic operations for such systems. The data model allows to associate with events e a set of possible time points T and with each time point $t \in T$ an interval for the probability that e occurred at t. We have presented explicit object base instances, where the sets of time points along with their probability intervals are simply enumerated, and implicit ones, where the sets of time points are expressed by constraints and their probability intervals by

probability distribution functions. Thus, implicit object base instances are succinct representations of explicit ones; they allow for an efficient implementation of algebraic operations, while their explicit counterparts make defining algebraic operations easy. We have defined our algebraic operations on both explicit and implicit object base instances, and shown that each operation on implicit object base instances correctly implements its counterpart on explicit instances.

There are numerous directions for future research. Building physical cost models and cost based query optimizers for TPOBs is a major challenge that must be addressed if applications such as the package and stock market example are to scale up for heavy duty use. Building mechanisms to update such databases poses yet another challenge. Building view creation and maintenance algorithms provides a third challenge. Developing an implementation of (the implicit version of) TPOBs poses a fourth major challenge as it will provide a testbed for all the algorithms resulting from the other problems mentioned here.

Appendix A: Table of Commonly Used Symbols in TPOB

Table 16.1: Notation

Symbol	Description	Section
$[\![\tau]\!]$	**atomic probabilistic type**	**9.3**
$\mathcal{C}$	**set of classes**	**10.1**
$\sigma(c)$	**type assignment of a class** c	**10.1**
$\Rightarrow$	**immediate subclass relationship**	**10.1**
$\Rightarrow^\star$	**subclass relationship (i.e., reflexive and transitive closure of** $\Rightarrow$**)**	**10.1**
$\mathrm{me}(c)$	**partition of the subclasses of a class** c	**10.1**
$\wp(c_1, c_2)$	**conditional probability that an object belongs to a class** c_1 **given that it belongs to an immediate superclass** c_2	**10.1**
$\mathbf{S} = (\mathcal{C}, \sigma, \mathrm{me}, \wp)$	**TPOB-schema**	**10.1**
ζ	**mapping from the set of classes to a set of objects**	**10.1**
$\pi(c)$	**set of object identifiers in a class** c	**10.3**
$\nu(o)$	**value of an object** $o \in \pi(c)$	**10.3**
$\mathbf{I} = (\pi, \nu)$	**TPOB-instance**	**10.3**
$\mathrm{ext(c)}(o)$	**probabilities with which an object** o **belongs to a class** c	**10.3**
$\pi^\star(c)$	**object identifiers in** c **and all subclasses of** c	**10.3**
$\sigma^\star(c)$	**inheritance completion of a type of a class** c	**10.2**
ε	**mapping from implicit values (instances) to explicit values (instances)**	**8.3**

Bibliography

[1] **S. Abiteboul. Querying semi-structured data. In *Proceedings of the ICDT*, pages 1–18, 1997.**

[2] **S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.**

[3] **S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.**

[4] **S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences*, 58(3):428–452, 1999.**

[5] **S. Adali and L. Pigaty. The darpa advanced logistics program. *To appear in Annals of Mathematics and Artificial Intelligence*, 2002.**

[6] **H. Almohamad and S.O.Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Transactions on pattern analysis and machine intelligence*, 15(5), 1993.**

[7] **M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Data Bases*, pages 53–64, 2000.**

[8] **E. I. Altman. Financial ratios, discriminant analysis, and the prediction of a corporate bankruptcy. *Journal of Finance*, pages 589 – 609, 1968.**

[9] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 497–508, 2001.

[10] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *Proceedings of the International Conference on Extending Database Technology*, 2002.

[11] M. Andries and G. Engels. Syntax and semantics of hybrid database languages. *Dagstuhl Seminar on Graph Transformations in Computer Science*, pages 19–36, 1993.

[12] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S.B. Zdonik. The object-oriented database system manifesto. In *Proceedings DOOD-89*, pages 40–57. Elsevier Science Publishers, 1989.

[13] R. Baeza-Yates. Algorithms for string searching: A survey. In *Proceedings of the SIGIR Forum*, volume 23, pages 34–58, 1989.

[14] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In Lecture Notes In Computer Science, editor, *Proceedings of the Combinatorial Pattern Matching*, pages 198–212, 1994.

[15] R. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[16] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[17] D. Barbosa, A. Barta, A. O. Mendelzon, G. A. Mihaila, F. Rizzolo, and P. Rodriguez-Guianolli. Tox - the toronto xml engine. In *Proceedings of the Workshop on Information Integration on the Web*, pages 66–73, 2001.

[18] Barnard chemical information ltd. http://www.bci1.demon.co.uk.

[19] H. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4:83–84, 1976.

[20] **S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu.** ***XQuery 1.0: An XML Query Language.available at http://www.w3.org/TR/xquery/*****.**

[21] **I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem.** ***Handbook of Combinatorial Optimization,*****, 4, 1999.**

[22] **P. A. Boncz, A. N. Wilschut, and M. L. Kersten. Flattening an object algebra to provide performance. In** ***Proceedings ICDE-98*****, pages 568–577. IEEE Computer Society, 1998.**

[23] **G. Boole.** ***The Laws of Thought*****. Macmillan, London, 1854.**

[24] **D. J. Bowersox, D. J. Class, and M. B. Cooper. Supply chain logistics management. Irwin/McGraw Hill, 2002.**

[25] **T. Bozkaya and M. Ozsoyogl. Indexing large metric spaces for similarity search queries.** ***ACM Transaction on Database Systems*****, 24(3):361–404, 1999.**

[26] **S. Brin. Near neighbor search in large metric spaces. In** ***Proceedings of the 21 International Conference on Very Large Data Bases*****, pages 574–584, Switzerland, 1995.**

[27] **V. Brusoni, L. Console, P. Terenziani, and B. Pernici. Extending temporal relational databases to deal with imprecise and qualitative temporal information. In** ***Recent Advances in Temporal Databases*****, pages 3–22. Springer, 1995.**

[28] **P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured and structured databases. In** ***Proceedings of the ACM SIGMOD SIGACT SIGART Symposium on Principles of Database Systems*****, pages 129–138, 1998.**

[29] **H. Bunke. Error correcting graph matching: On the influence of the underlying cost function.** ***IEEE Transactions on Pattern Analysis and Machine Intelligence*****, 21(9):917–922, 1999.**

[30] H. Bunke. Recent developments in graph matching. In *Proceedings of the ICPR*, pages 2117–2124, 2000.

[31] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[32] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, pages 245–253, 1983.

[33] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *SSPR/SPR*, pages 123–132, 2002.

[34] H. Bunke and K. Shearer. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, pages 689–694, 1997.

[35] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communication of ACM*, pages 230–236, April 1973.

[36] J.B. Burns and E.M. Riseman. Matching complex images to multiple 3d objects using view description networks. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 328–334, 1992.

[37] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106:21–60, 1992.

[38] J. B. Caouette, E. I. Altman, and P. Narayanan. *Managing credit risk: The next great financial challenge*. John Whiley, 1998.

[39] D. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th Annual CM Symposium on Principles of Programming Languages,*, pages 168–177, 1987.

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

[41] S. 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, 1997.

[42] S. 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, 1996.

[43] T. Chiueh. Content-based image indexing. In *Proceedings of the 20 International Conference on Very Large Data Bases*, pages 582–593, 1994.

[44] W. J. Christmas, J. Kittler, and M. Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):749–764, 1995.

[45] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23th International Conference on Very Large Data Bases*, pages 426–435, 1997.

[46] J. Clark and S. DeRose. *Xml Path Language (Xpath)*. http://www.w3.org/TR/xpath, 1999.

[47] R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized o(n log 3 m)-time. In *Proc. of the 29th Annual ACM Symp. Theory of Computing*, pages 66–75, 1997.

[48] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic o(n log 3 n)-time. In *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 245–254, 1999.

[49] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Artificial Intelligence Research*, 1:231–255, 1994.

[50] L.P. Cordella, P. Foggia, C. Sansone, F. Tortorella, and M. Vento. Graph matching: A fast algorithm and its evaluation. In *Proc. of the 14th International Conference on Pattern Recognition*, pages 1582–1584, 1998.

[51] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An efficient algorithm for the inexact matching of arg graphs using a contextual transformational model. In *Proceedings of the 13th ICPR*, volume 3, pages 180–184. IEEE Computer Society Press, 1996.

[52] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, 2001.

[53] D. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *ACM*, 17(1):51–64, 1970.

[54] Y. F. Day, S. Dagtas, M. Iino, A. A. Khokhar, and A. Ghafoor. An object-oriented conceptual modeling of video data. In *Proceedings ICDE-95*, pages 401–408, 1995.

[55] Y. F. Day, S. Dagtas, M. Iino, A. A. Khokhar, and A. Ghafoor. Spatio-temporal modeling of video data for on-line object-oriented query processing. In *Proceedings ICMCS-95*, pages 98–105, 1995.

[56] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36, 1998.

[57] A. Dekhtyar, R. Ross, and V. S. Subrahmanian. Probabilistic temporal databases, I: Algebra. *ACM Transactions on Database Systems*, 26(1):41–95, March 2001. More detailed version: Technical Report CS-TR-3987, University of Maryland, February 1999, available at `http://www.cs.umd.edu/~dekhtyar/academ/publications/tp-final.ps`.

[58] **F. Bancilhon C. Delobel and P. Kanellakis.** ***Building an Object-Oriented Database System: The Story of O_2*****. Morgan Kaufmann, 1991.**

[59] **A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for xml.** ***Journal of Computer Networks*****, 31(11-16):1155–1169, 1999.**

[60] **D.J. Dewitt, N. Kabra, J. Luo, J. Patel, and J. Yu. Client-server paradise.** ***Proceedings of the 20 International Conference on Very Large Data Bases*****, pages 558–569, 1994.**

[61] **Y. Dinitz, A. Itai, and M. Rodeh. On an algorithm of zemlyachenko for subtree isomorphism.** ***Inform. Process. Lett.*****, 70(3):141–146, 1999.**

[62] **S. Djoko, D. J. Cook, and L. B. Holder. An empirical study of domain knowledge and its benefits to substructure discovery.** ***IEEE Transactions on Knowledge and Data Engineering*****, 9(4), 1997.**

[63] **M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching.** ***JACM*****, 14(2):205–213, 1994.**

[64] **P. Dublish. Some comments on the subtree isomorphism problem for ordered trees.** ***Inform. Process. Lett.*****, 36(5):273–275, 1990.**

[65] **D. Dubois and H. Prade. Processing fuzzy temporal knowledge.** ***IEEE Transactions on Systems, Man, and Cybernetics*****, 19(4):729–744, 1989.**

[66] **S. Dutta. Generalized events in temporal databases. In** ***Proceedings of the 5th International Conference on Data Engineering (ICDE-89)*****, pages 118–126. IEEE Computer Society, 1989.**

[67] **C. Dyreson and R. Snodgrass. Supporting valid-time indeterminacy.** ***ACM Transactions on Database Systems*****, 23(1):1–57, 1998.**

[68] **T. Eiter, J. J. Lu, T. Lukasiewicz, and V. S. Subrahmanian. Probabilistic object bases.** ***ACM Transactions on Database Systems*****, 26(3):264–312, September 2001.**

[69] T. Eiter, T. Lukasiewicz, and M. Walter. A data model and algebra for probabilistic complex values. *Annals of Mathematics and Artificial Intelligence*, 33(2-4):205–252, December 2001.

[70] G. Engels, C. Lewerentz, M. Nagl, W. Schfer, and A. Schrr. Building integrated software development environments part i: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[71] M. A. Eshera and K.-S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems Man and Cybernetics*, 14(3):353–363, 1984.

[72] F.Cesarini, M.Lastri, S.Marinai, and G.Soda. Page classification for meta-data extraction from digital collections. In *Proceedings of the DEXA, Munich*, pages 82–91, 2001.

[73] J. Feng, M. Laumy, and M. Dhome. *Inexact Matching using Neural Networks*. E. Gelsema and e. L.N. Kanal, Pattern recognition in practice IV: Multiple Paradigms, Comparative Studies and Hybrid Systems,North–Holland, 1994.

[74] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 414–425, 1998.

[75] M.-L. Fernandez and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, pages 753–758, 2001.

[76] A. Ferro, G. Gallo, and R. Giugno. Error-tolerant retrieval for structured images. In Lecture Notes In Computer Science, editor, *VISUAL*, volume 1614, pages 51–59, 1999.

[77] A. Ferro, G. Gallo, R. Giugno, and A. Pulvirenti. Best-match retrieval for structured images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(7):707–718, 2001.

[78] **P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking.** ***Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*****, pages 176–187, 2001.**

[79] **P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In** ***Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*****, pages 188–199, 2001.**

[80] **S. Fortin. The graph isomorphism problem. Technical Report 96–20, University of Alberta, Edomonton, Alberta, Canada, 1996. Technical Report.**

[81] **W. B. Frakes and R. Baeza-Yates.** ***Information Retrieval: Data Structures and Algorithms*****. Prentice Hall, 1992.**

[82] **S. Gadia, S. Nair, and Y. C. Poon. Incomplete information in relational temporal databases. In** ***Proceedings of the 18th International Conference on Very Large Databases*****, pages 395–406, 1992.**

[83] **M. Garey and D. Johnson.** ***Computers and Intractability: A Guide to the Theory of NP-Completeness*****. Freeman and Company, 1979.**

[84] **S. Gold and A. Rangarajan. A graduated assignment algorithm for graph matching.** ***IEEE Transactions on Pattern Analysis and Machine Intelligence*****, 18(4):377–388, 1996.**

[85] **R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In** ***Proceedings of the International Conference on Very Large Data Bases*****, pages 436–445, 1997.**

[86] **G. H. Gonnet and F. W. Tompa. Mind your grammar: A new approach to modeling text. In** ***Proceedings of the International Conference on Very Large Data Bases*****, pages 339–346, 1987.**

[87] **R. Grossi. Further comments on the subtree isomorphism for ordered trees.** ***Inform. Process. Lett.*****, 40(5):255–256, 1991.**

[88] R. Grossi. A note on the subtree isomorphism for ordered trees and related problems. *Inform. Process. Lett.*, 39(2):81–84, 1991.

[89] R. Grossi. On finding common subtrees. *Theor. Comput. Sci.*, 108(2):345–356, 1993.

[90] A. Gupta and N. Nishimura. Finding largest subtrees and smallest supertrees. *Algorithmica*, 21(2):183–210, 1998.

[91] R. H. Guting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 297–308, 1994.

[92] K. Hirata and T. Kato. Query by visual example content based image retrieval. In *Proceedings of the Advances in Database Technology*, pages 56–71, 1992.

[93] A. Hlaoui and S. Wang. A new algorithm for inexact graph matching. In *Proceedings of the 16th ICPR*, 2002.

[94] C. M. Hoffman and M. J. O'Donnell. Pattern matching in trees. *ACM*, 29(1):68–95, 1982.

[95] P. Hong and T.S. Huang. Spatial pattern discovering by learning the isomorphic subgraph from multiple attributed relational graphs. *Proceedings of ENTCS*, 2001.

[96] J. Hopcroft and J. Wong. Linear time algorithm for isomorphism of planar graphs. *Proceedings of the Sixth Annual ACM Symp. on Theory of Computing*, pages 172–184, 1974.

[97] C. A. James, D. Weininger, and J.Delany. *Daylight theory manual-Daylight 4.71*. Daylight Chemical Information Systems, www.daylight.com, 2000.

[98] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.

[99] **J. M. Jolion. Graph matching: What are we talking about?** ***Pattern Recognition Letters*****, 2002.**

[100] **Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In** ***Proceedings of the Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*****. ACM, 2001.**

[101] **T. Kato, T. Kurita, N.Otsu, and K.Hirata. A sketch retrieval method for full color image databasequery by visual example. In** ***Proceedings of the 11th IAPR International Conference on Pattern Recognition*****, pages 530–533, 1992.**

[102] **B. Kelley.** ***Frowns*****. http://staffa.wi.mit.edu/people/kelley/, 2002.**

[103] **Brian Kelley. Frowns. http://staffa.wi.mit.edu/people/kelley/, 2002.**

[104] **P. Kilpelainen. Tree matching problems with applications to structured text databases. Technical Report Report A-1992-6, University of Helsinki, Finland, November 1992.**

[105] **P. Kilpelainen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In** ***Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*****, pages 214–222, 1993.**

[106] **P. Kilpelainen and H. Mannila. Query primitives for tree-structured data. In** ***Proceedings of the Annual Symposium on Combinatorial Pattern Matching*****, pages 213–225, 1994.**

[107] **P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion.** ***SIAM J. Comput*****, 24(2):340–356, 1995.**

[108] **A. Kitamoto, C. Zhou, and M. Takagi. Similarity retrieval of noaa satellite imagery by graph matching. In** ***Proceedings of the SPIE*****, pages 60–73, 1993.**

[109] **D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings.** ***SIAM Journal on Computing*****, 6(2):323–350, 1977.**

[110] Y. Kornatzky and S. E. Shimony. A probabilistic object-oriented data model. *Data & Knowledge Engineering*, 12:143–166, 1994.

[111] Y. Kornatzky and S. E. Shimony. A probabilistic spatial data model. *Information Sciences*, 90:51–74, 1996.

[112] S. R. Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th annual IEEE Symposium on Foundations of Computer Science*, pages 178–183, 1992.

[113] M. Koubarakis. Complexity results for first order theories of temporal constraints. In *Proceeding of the 4th International Conference on Knowledge Representation and Reasoning (KR-94)*, pages 379–390, 1994.

[114] M. Koubarakis. Database models for infinite and indefinite temporal information. *Information Systems*, 19(2):141–173, 1994.

[115] S. Kraus, Y. Sagiv, and V. S. Subrahmanian. Representing and integrating multiple calendars. Technical Report CS-TR-3751, University of Maryland, 1996.

[116] H. E. Kyburg, Jr. Interval-valued probabilities. In G. de Cooman, P. Walley, and Fabio G. Cozman, editors, *Imprecise Probabilities Project*. 1998. Available from `http://ippserv.rug.ac.be/`**.**

[117] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. ProbView: A flexible probabilistic database system. *ACM Transactions on Database Systems*, 22(3):419–469, 1997.

[118] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik. The aqua data model and algebra. In *Proceedings of the 4th Workshop on Database Programming Languages*, pages 157–175, 1993.

[119] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Journal of Calcols 9*, pages 341–354, 1972.

[120] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 361–370, 2001.

[121] J. Llads, E. Mart, and J.J. Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1137–1143, 2001.

[122] S.Y. Lu. A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1:219–224, 1979.

[123] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer System Science*, pages 42–65, 1982.

[124] E. Makinen. On the subtree isomorphism problem for ordered trees. *Inform. Process. Lett.*, 32(5):271–273, 1989.

[125] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[126] N. Manocha, D. J. Cook, and L. B. Holder. Structural web search using a graph-based discovery system. In *Proceedings of the Florida Artificial Intelligence Research Symposium*, 2001.

[127] D. W. Matula. Subtree isomorphism in $o(n^{5/2})$. *Ann. Discrete Math.*, 2:91–106, 1978.

[128] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. In *Proceedings of the SIGMOD*, volume 26, pages 54–66, September 1997.

[129] B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, pages 45–87, 1981.

[130] B. T. Messmer. ***Efficient Graph Matching Algorithms for Preprocessed Model Graphs*****. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Bern, 1996.**

[131] B. T. Messmer and H. Bunke. Subgraph isomorphism detection in polynominal time on preprocessed model graphs. In ***Proceedings of ACCV*****, pages 373–382, 1995.**

[132] B. T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: a decomposition approach. ***IEEE Trans. on Knowledge and Data Engineering*****, 12(2):307–323, 2000.**

[133] G. Miklau and D. Suciu. Containment and equivalence of xpath expressions. In ***Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*****, 2002.**

[134] T. Milo and D. Suciu. Index structures for path expressions. In ***Proceedings of the International Conference on Database Theory*****, pages 277–295, 1999.**

[135] R. Myers, R.C. Wilson, and E. R. Hancock. Bayesian graph edit distance. In ***10th Int. Conf. on Image Analysis and Processing, IEEE*****, 1998.**

[136] G. Nagy and S. Seth. Hierarchical representation of optically scanned documents. In ***Proceedings of the ICPR*****, pages 347–349, 1984.**

[137] National cancer institute. http://www.nci.nih.gov/.

[138] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The qbic project: Querying images by content using color, texture, and shape. in storage and retrieval for image and video databases. In ***Proceedings of the SPIE*****, volume 1908, pages 173–187, 1993.**

[139] G. Gallo S. Garraffo R. Giugno A. Nicolosi. The lost follis. ***Proceeding of Eurographics*****, 1999.**

[140] N.J. Nilsson. ***Principles of artificial intelligence*****. Tioga, Palo Alto CA, 1980.**

[141] S. Nirenburg, S. Beale, and C. Domashnev. A full-text experiment in example-based machine translation. In ***Proceedings of the Int'l Conf. New Methods in Language Processing, Manchester*****, pages 78–87, 1994.**

[142] K. Oflazer. Error-tolerant retrieval of trees. ***IEEE Transactions on Pattern Analysis and Machine Intelligence*****, 19(12), 1997.**

[143] Gultekin Özsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. ***IEEE Transactions on Knowledge and Data Engineering*****, 7(4):513–532, 1995.**

[144] J. G. Pailloncy, A. Deruyver, and J. M. Jolion. From pixels to predicates revisited in the graphs framework. In ***2rd Int. Workshop on Graph based Representations,GbR99*****, 1999.**

[145] T. Pavlidis. Representation of figures by labeled graphs. ***Pattern Recognition*****, 4(1):5–17, 1972.**

[146] Protein data bank. http://www.rcsb.org/pdb/.

[147] S. W. Reyner. An analysis of a good algorithm for the subtree problem. ***SIAM J. Comput.*****, 6(4):730–732, 1977.**

[148] S. Ross. ***A First Course in Probability*****. Prentice Hall, 2001.**

[149] S. Safo and M. Nagao. Towards memory-based translation. In ***Proceedings of the 13th Int'l Conf. Computational Linguistics*****, volume 3, pages 247–252, 1990.**

[150] H. Schek and P. Pistor. Data structures for an integrated data base management and information retrieval system. In ***Proceedings of the 8th International Conference on Very Large Data Bases*****, pages 197–207. Morgan Kaufmann, 1982.**

[151] T. Schlieder and F. Naumann. Approximate tree embedding for querying xml data. In ***Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval*****, Greece, 2000.**

[152] D.C. Schmidt and L.E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. ***Association for Computing Machinery*****, 23:433–445, 1976.**

[153] A. Schrr. Logic based structure rewriting systems. ***Graph Transformations in Computer Science*****, pages 341–357, 1993.**

[154] S.M. Selkow. The tree-to-tree editing problem. ***Information Processing Letters*****, 6:184–186, 1977.**

[155] D.S. Seong, H.S. Kim, and K.H. Park. Incremental clustering of attributed graphs. ***IEEE Transactions on System, Man, and Cybernetics*****, 23(5):1399–1411, 1993.**

[156] R. Shamir and D. Tsur. Faster subtree isomorphism. ***J. Algorithms*****, 33(2):267–280, 1999.**

[157] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In ***Proceedings of the 25 International Conference on Very Large Data Bases*****, pages 302–314, 1999.**

[158] L.G. Shapiro and R.M. Haralick. Organization of relational models for scene analysis. ***IEEE Transactions on Pattern Analysis and Machine Intelligence*****, pages 595–602, 1982.**

[159] D. Sharvit, J. Chan, H. Tek, and B. B. Kimia. Symmetry-based indexing of image databases. In ***FYI: CVPR'98 Workshop on Content-Based Access of Image/Video*****, 1998.**

[160] D. Shasha and R. Giugno. Graphgrep: A fast and universal method for querying graphs. ***Proceeding of the International Conference in Pattern recognition (ICPR)*****, 2002.**

[161] D. Shasha and J. T. L. Wang. New techniques for best-match retrieval. ***ACM Transaction on Information Systems*****, 8(2):140–158, 1990.**

[162] D. Shasha, J. T. L. Wang, H. Shan, and K. Zhang. Atreegrep: Approximate searching in unordered trees. Submitted, 2002.

[163] D. Shasha, J. T. L. Wang, K. Zhang, and F. Y. Shih. Pattern matching in unordered trees. In *Proceedings of the ICTAI*, pages 352–361, 1992.

[164] D. Shasha, J.T-L Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. *Proceeding of the ACM Symposium on Principles of Database Systems (PODS)*, pages 39–52, 2002.

[165] G. M. Shaw and S. B. Zdonik. A query algebra for object-oriented databases. In *Proceedings ICDE-90*, pages 154–162. IEEE Computer Society, 1990.

[166] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *Proceedings of the ICDE*, pages 572–581, 1999.

[167] R. T. Snodgrass. *Monitoring Distributed Systems: A Relational Approach*. PhD thesis, Carnegie Mellon University, 1982.

[168] R. T. Snodgrass. Temporal object-oriented databases: A critical comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*. ACM Press and Addison-Wesley, 1995.

[169] H. Sossa and R. Horaud. Model indexing: The graph-hashing approach. In *Proceedings of Computer Vision and Pattern Recognition*, pages 811–814, 1992.

[170] H. Su, H. Kuno, and E. Rundensteiner. Automating the transformation of xml documents. In *Proceedings of the Workshop on Web Information and Data Management*, 2001.

[171] B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik. The aqua approach to querying lists and trees in object-oriented databases. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 80–89, 1995.

[172] **B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik. The AQUA approach to querying lists and trees in object-oriented databases. In *Proceedings ICDE-95*, pages 80–89. IEEE Computer Society, 1995.**

[173] **B. Subramanian, S. B. Zdonik, T. W. Leung, and S. L. Vandenberg. Ordered types in the aqua data model. In *Proceedings of the 4th Workshop on Database Programming Languages*, pages 115–135, 1993.**

[174] **D. Suciu. An overview of semistructured data. *Proceedings of the SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29, 1998.**

[175] **Dan Suciu and Jan Paredaens. Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure. In *Proceedings PODS-94*, pages 201–209. ACM Press, 1994.**

[176] **K. C. Tai. The tree-to-tree correction problem. *Journal of ACM*, 26:422–433, 1979.**

[177] **A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1994.**

[178] **M. Thorup. Efficient preprocessing of simple binary pattern forests. *J. Algorithms*, 20(3):602–612, 1996.**

[179] **W. H. Tsai and K. S. Fu. Error-correcting isomorphism of attributed relational graphs for pattern analysis. *IEEE Transactions on Systems Man and Cybernetics*, 9:757–768, 1979.**

[180] **J.K. Uhlmann. Satisfying general proximity/similarity queries with metric. *Information Processing Letters*, 40:175–179, 1991.**

[181] **J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23:31–42, 1976.**

[182] S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 10(5):695–703, 1988.

[183] G. Valiente. An efficient bottom-up distance between trees. In *Proc. of the 8th Int. Symp. String Processing and Information Retrieval*, pages 212–219, 2001.

[184] Scott L. Vandenberg and David J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proceedings SIGMOD-91*, pages 158–167. ACM Press, 1991.

[185] R. M. Verma. Strings, trees, and patterns. *Inform. Process. Lett.*, 41(3):151–161, 1992.

[186] R. M. Verma and S. W. Reyner. An analysis of a good algorithm for the subtree problem,corrected. *SIAM J. Comput.*, 18(5):906–908, 1989.

[187] V. Vianu. A web odyssey: from Codd to XML. In *Proceedings of the Symposium on Principles of Database Systems*, 2001.

[188] J. T. L. Wang, B. A. Shapiro, and D. Shasha. *Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications*. Oxford University Press, New York, 1999.

[189] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and C.-Y. Chang. Automated discovery of active motifs in multiple rna secondary structures. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 70–75, 1996.

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

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

[192] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, S. Dikshitulu, I. Rigoutsos, and K. Zhang. Automated discovery of active motifs in three dimensional molecules. In ***Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*****, pages 89–95, 1997.**

[193] D. Weininger. Smiles. introduction and encoding rules. ***Journal of Chemical Information in Computer Science*****, 28(31), 1988.**

[194] R. Wilson and E. Hancock. Bayesian compatibility model for graph matching. ***Pattern Recognition Letters*****, 17:263–276, 1996.**

[195] A. Wong and M. You. Entropy and distance of random graphs with application to structural pattern recognition. ***IEEE Transactions Pattern Analysis and Machine Intelligence*****, 7(5):599–609, 1985.**

[196] S. Wu and U. Manber. Fast text searching allowing errors. ***Communications of the ACM*****, 35(10):83–91, 1992.**

[197] Extensible Markup Language (XML). http://www.w3.org/xml/.

[198] Xml treediff. http://www.alphaworks.ibm.com/aw.nsf/faqs/xmltreediff.

[199] M. Yannakakis. Graph theoretic methods in database theory. In ***Proceedings of the 9th ACM Symp. on Principles of Database Systems*****, pages 230–242, 1990. invited paper.**

[200] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. ***Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*****, pages 311–321, 1993.**

[201] K. Zhang, D. Shasha, and J. T. Wang. Approximate tree matching in the presence of variable length don't cares. ***Journal of Algorithms*****, 16(1):33–66, 1994.**

[202] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. ***Information Processing Letters*****, 42:133–139, 1992.**