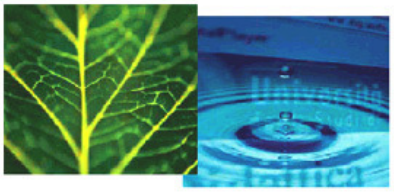
PhD Dissertation



International Doctoral School in Information and Communication Technologies

DISI - University of Trento

Advanced Query Paradigms for the Novice User

Davide Mottin

Advisor:
Prof. Yannis Velegrakis
University of Trento

Co-Advisor:
Prof. Themis Palpanans
University of Paris Descartes

April 2015

"A worker may be the hammer's master, but the hammer still prevails. A tool knows exactly how it is meant to be handled, while the user of the tool can only have an approximate idea."

Milan Kundera

Abstract

Query answering is one of the most important processes in search systems, for it connects users to the information stored in data sources. A query is a set of specifications or constraints that the user provides to describe the objects of interest. As such, answering a query means retrieving those objects from the data source that match the user constraints.

An answer from a search system may not fully satisfy the user. This happens if the answer does not contain the required object or it contains a number of irrelevant results. Commonly, the user does not know how to describe the query and ends up with one overly generic or specific or, even worse, she is not even aware of the correct conditions to describe the expected results. These problems are particularly evident when the database is interrogated by a novice user who, by definition, does not have sufficient technological skills to understand complicated query languages, or simply gives up if the system does not respond properly or timely.

In this dissertation, we focus on three common problems in the broad query answering process to help novice users find the correct answers when the system does not provide sufficient support. First, we look at the empty answer problem, where the user provides a very specific query for which no answer exists in the database. In particular, we concentrate on interactive approaches for novice users. We end up with a rich probabilistic framework that includes user preferences and smoothly guides the user towards the most likely answers by means of simple yes/no questions on the query conditions to be discarded. Second, we analyze the information overload problem, that is complementary to the first. In this case the user provides a too generic query that returns a large set of potentially irrelevant results. We

tackle the problem in structured databases, and more specifically, labeled graphs. The solution we propose returns a set of refinements (i.e., more specific queries) of the input query that, once executed, covers all the initial results. Third, we propose and study a completely novel query paradigm that assumes that the user is not able to describe the query conditions to retrieve the objects of interest. In this regard, we introduce exemplar queries, that allow the user to specify a single element in the result set and let the system infer the others. We provide clear semantics and a solution that works in large knowledge graphs.

Finally, we validate the solutions for the three problems both in terms of theoretical results and experimental evaluation and we prove that the proposed methods efficiently scale up to very large real datasets.

Keywords

Database usability, query answering, novice users, query reformulation

Contents

List of Figures

List of Tables

Basic Notation

$\mathcal{D}$	database
Q	query
Q'	reformulated query
$\mathcal{T}$	relaxation/reformulation tree
$\mathcal{D}_Q$	result set of query Q
$\mathcal{L}$	infinite set of labels
$\mathcal{V}$	infinite set of attribute values
$\mathcal{O}$	infinite set of object identifiers

Relational

$(v_1, ..., v_n)$	tuple with n values
Dom_A	domain of attribute A

Graphs

$G : \langle N, E \rangle$	graph G with node set N and edge set $E \subseteq N \times N$
N	graph node set
E	graph edge set
$u - v$	not directed edge between nodes u and v
$u \rightarrow v$	directed edge from node u to v
$u \overset{\ell}{-} v$	not directed edge labeled ℓ between nodes u and v
$u \overset{\ell}{\rightarrow} v$	directed edge labeled ℓ from node u to v

Dedicated to my parents from where my life started
and to God to where my life will end.

Chapter 1

Introduction

The increasing number of people accessing information through search engines and database systems has nurtured a considerable interest in the exploration of simpler and more user-centric ways of accessing the data. This aspect is particularly important when the user is not a system or a data expert, what is commonly referred to as a *novice user*. Novice users, which are the vast majority of people using computers [Zic13] are, for their intrinsic nature, less inclined to access data for which the system provides little or no help at all. It is also true that a considerable amount of people do not approach basic search systems, such as search engines, because they are either too difficult, or they hardly match their expectations [Zic13, TS05], contributing to the phenomenon of digital divide [Fri14].

The preferred way to organize and access the data is through softwares called database management systems, in short databases. Data in a database are represented and stored in a plethora of different data-models depending on the use, such as relational [Cod70], object-oriented [SM95], semi-structured [Abi97], and graphs [Kun90], among others. In particular, we focus on graphs and relational databases, while still supplying a broad range of applications, from online e-commerce websites to knowledge bases.

The design of database systems is based on the principles of abstracting the underlying storage layer and providing fast access to the data. While from the efficiency standpoint the systems are constantly improved to have timely answers even with large amount of data, database ease of use remains, in some cases, limited [Dat83]. A usable system provides assistance in all the phases of the information retrieval process,

from the formulation of the query to the presentation of the results. For this reason, much effort has been devoted to the study of usable solutions [JCE+07], such as keyword-based interfaces [YQC09], query result explanation [Mot84], interactive approaches [MK09, BRWD+08], and data-exploration tools [sql, ora]. The steps towards the design of fully usable databases involve more natural search capabilities that explicitly take into account the user in the process.

Searching capabilities over databases are enabled by queries. A query is a set of conditions that describes the object of interest, and is commonly specified in a database-specific language. Queries may fail to accomplish user needs for many reasons. For instance, a query may be too stringent and return no answers (empty answer problem) [MK09, Jan07, Mot84] or too generic and return too many, potentially irrelevant, answers (information overload problem) [BRWD+08, BBCV11, ZTR13]. These two common issues have been recently studied and solutions have been proposed to assist the user in the process of progressively refining the query to find the desired results. However, if expert users can find appropriate workarounds, novice users need a complete assistance by the system [TS05]. Interactivity and result diversification are two important factors that can make a big difference for a novice user. The former refers to the design of systems that progressively propose better queries, while the latter to the presentation of results or queries that are able to describe the results with short summaries.

Another issue that makes a database less palatable to novice users is the rigidity of the query languages. These languages use complex operators and syntax, diminishing the usability of the system itself. This problem has been traditionally solved with keyword-based interfaces [YQC09], at the price of introducing ambiguity in the interpretation of the query. An orthogonal approach would be to change the query paradigm in order to find objects similar to the one the user has in mind and let the system infer the rest. This is a compromise between the ambiguity of keyword-based interfaces and the rigidity of the formal query languages.

This dissertation discusses challenges and solutions to improve the interaction with database systems for novice users when facing incomplete or missing results, irrelevant results, or hard to express queries, and proposes novel query paradigms that treat usability as a first class citizen in query answering. The rest of this chapter is organized as follows. Section 1.1 presents the motivations behind this work, identifying the main

niches that constitute the research challenges in Section 1.2. Examples of applications of the techniques presented here are discussed in Section 1.3. Section 1.4 concludes with an outline of the thesis.

1.1 Motivation

Users, looking for specific answers, use search systems to retrieve the required information. However, from what a user expects as answer and what the database returns there is often an asymmetry. The database contains a set of objects that, in most cases, do not entirely correspond to the user needs.

In order to access the data, the user poses a query to the database. A query is commonly represented as a set of constraints over the items to be returned. The database engine then returns the items matching the query constraints, that might be empty if no such item exists. The entire process is called *query-answering*.

Unfortunately, a novice user needs far more assistance in query answering than normal users. A novice user typically: -uses an imprecise or limited vocabulary, or at least she is unable to formulate the correct query; -is not fully capable of describing the required information; -is not willing to wait for answers and expects guidance and explanations from the system. Therefore, the study of systematic ways to assist novice users in answering queries is of paramount importance for the database community. Moreover, as stated by other researchers [JCE+07], a system that provides little interaction with the user and no explanation of the results falls short to meet user expectations.

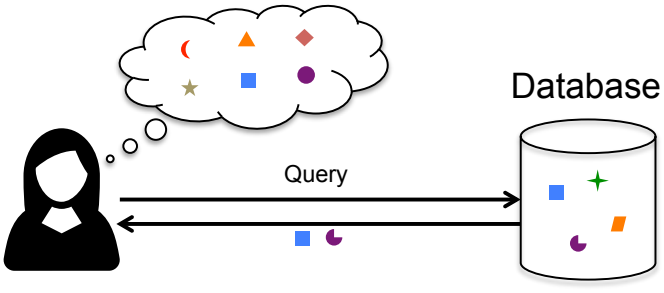


Figure 1.1: The user perspective of a database search.

This work focuses on methods to help the user find her expected results with little or no knowledge about the database and its query languages. Consider the situation depicted in Figure 1.1. The user is looking for some specific results, and asks a database, in which

she thinks to find the desired information, to provide them. Unfortunately, the database does not always respond to the expectations, since, in most cases, simply it does not contain the specific objects the user is looking for.

The user might encounter the following issues, among others. First, the set of results of the query is empty, and the database needs to guide the user towards the desired answer. This is referred to as the *empty answer* problem, in which the system proposes to the user a set of relaxed queries with less conditions than the ones she issued. Second, the user is looking for some structure in the data, but, since she does not know how to specify the query, she provides a too general description that returns a big number of irrelevant results and expects further help in defining the query. In database community, this is referred to as the *information overload* problem and the set of queries proposed to the user in this case consists in more specific queries in order to restrict the number of results. Third, the user is not able to correctly express the desired information need but she knows one of the items in the result set. We call this problem *inexpressible queries*, referring to the inability of the user of defining the set of constraints needed to retrieve the objects of interest.

The following section discusses in more detail these three challenges and the solutions proposed in this work.

1.2 Research Challenges and Contributions

Dealing with novice users opens a number of research questions that require ad-hoc solutions. This dissertation addresses three important aspects in the design of databases for novice users, namely *empty answer*, *information overload*, and *inexpressible queries*.

Empty Answer

Users searching for specific information in a database may over-specify the conditions in the query, and find no item in the data source satisfying all the provided conditions at once. This is commonly referred to as the *empty-answer problem*. A popular way to cope with empty-answers is *query relaxation*, which attempts to reformulate the original query into a new query, by removing (i.e., relaxing) some conditions, so that the result of the new query is likely to contain items similar to those searched.

Although query relaxation is typically *non-interactive* [Cha90, Jan07, Jun04], meaning that all the possible relaxations are returned to the user at once, in this work we propose an interactive approach [MMBR$^+$13, MMBR$^+$15]. The user starts from an empty-answer query and the system presents a single relaxation together with a "yes/no" choice of relaxing the query. Each step slightly perturbs the query, until it returns either a non-empty answer, or any further change in the query conditions does not result into a non-empty answer.

In this dissertation, we propose a novel principled interactive framework that optimizes a wide variety of application-dependent objective functions. For instance, the user might be suggested steps to find the cheapest car or, conversely the seller may decide to steer the relaxations towards more expensive automobiles. The framework might also be used to find the most similar items to those expressed in the query. User preferences and beliefs are modeled as probability functions and embedded in the interactive process, for which the framework computes a probability of saying "yes" (accept) or "no" (reject) the relaxation proposed by the system.

List of publications. In this topic we contributed with the following publications. The interactive optimization framework for empty-answer has appeared in [MMBR$^+$13] and extended with top-k relaxations and cardinality constraints on the size of the results in [MMBR$^+$15]. We also proposed a system demonstration in [MMBR$^+$14].

Information overload

In large databases, the number of items responding to a single query may be incredibly large. This is a typical issue that affects novice users, forcing them to either formulate a more specific query or search the items of interest in the huge list of results. This problem is usually mentioned as *information overload* or *many-answers problem* [MK09], and is particularly problematic as it requires the user to further inspect the results either manually or by repeatedly interacting with the database.

This dissertation proposes a solution for the information overload problem when the database is a set of structured objects and the query is a pattern to be found [MBG15]. A database containing a set of structures is commonly referred to as a *graph database*. The solution is based on a principled query refinement approach, in which the query is considered as a first-class citizen in the structure search. While query reformulation has been studied in various contexts, such as relational databases [MK09], keyword search

on structured data [TY09], and web search [HE09], to the best of our knowledge, this is the first work dealing with graph databases.

Dealing with structures conveys a number of performance issues, mainly because the generic problem of searching structures in a graph is **NP**-complete. Therefore, a thorough study of the problem and efficient solutions are proposed here. We base the formulation of our problem on the assumption that the reformulations produced should provide an effective high-level description of the original query results. In line with this intuition, we formalize the query-reformulation problem in graph databases as the problem of finding a set of k reformulations that maximize a linear combination of the *coverage* of the original query results and the *diversity* among the reformulations. This is modeled as an optimization problem that can be reduced to the MAX-SUM DIVERSIFICATION problem and, based on a recent theoretical result [BLY12], we propose a greedy $\frac{1}{2}$-approximated algorithm. We describe efficient algorithms to solve the key step of the greedy algorithm, i.e., finding the reformulation leading to the maximum increment of the objective function, that unfortunately relies on a #**P**-complete problem. Our goal is to solve such a step efficiently *while still guaranteeing optimality*, as this is needed to preserve the aforementioned approximation guarantee.

List of publications. Graph query reformulation with diversity has been submitted for publication in a conference [MBG15].

Inexpressible queries

The hidden assumption behind query-answering is that the user is aware of the characteristics of the structures of interest and can (at least partially) describe them in the query. However, this is not always as easy as it might seem at first. There are cases in which a user knows one single element among the desired results and expects the system to return the others. In all these cases, traditional query-answering fails its mission of helping the user in the process.

In this thesis, we propose *exemplar queries*, a novel query paradigm that treats the user query as a representative of the results to be returned. We define the novel problem of *exemplar query answering* and show its broad range of applicability, from search engines to databases. Exemplar queries can form the basis of a new generation of search engines that use them as the main query evaluation mechanism, or they can be used to enhance the services that existing search engines are currently offering. Note that this approach

is different from query relaxation [MK09, MMBR$^+$13], which aims at producing more generic versions of a query.

This work proposes a successful application of this paradigm to large knowledge graphs (i.e., graphs where nodes are real world objects/concepts and edges are relationships between them). A query in a knowledge graph is a substructure of the graph while the answers to an exemplar query are substructures similar to it. Finding similar substructures in a graph is, in most cases, a hard problem. In this dissertation, we show how we are able to efficiently find solutions to an exemplar query on large knowledge graphs and we propose a working prototype [MLVP14b] that exploits the entire Freebase [1] as knowledge graph for real-time answers. Exact and approximate solutions are presented as well as two different similarity functions to compute the results, namely subgraph isomorphism and strong simulation [Val79, MCF$^+$14].

List of publications. In this topic we contributed with the following publications. Exemplar queries have been first introduced in a seminal work, in which we also presented the algorithms to find isomorphic results [MLVP14a]. The extension with a different similarity function to retrieve results, namely strong simulation, have been recently submitted to a journal [MLVP15]. Moreover, we presented a system prototype [MLVP14b] and a vision paper to describe the interesting directions that we intend to explore, apart from nurturing further research [LMP$^+$14].

1.3 Applications

Exploratory Data Analysis

The three methods described in this dissertation can be employed in exploratory data analysis [Tuk77, ORPF13]. Exploratory data analysis is the process of finding patterns, regularities or distributional properties in the data with the purpose of understanding whether some information can be extracted from. Take for instance a lawyer that is interested in lawsuits about arsons in Alaska. After loading the information into a database the analyst perform some exploratory search. Our methods can greatly help her job. Assume the database contains cases of legal actions in California and Michigan but not Alaska. In this case, since no results are returned the empty-answer framework can guide her toward the most reasonable relaxation. On the other hand, if there are

[1] `http://www.freebase.com`

too many such lawsuits, then query reformulation is used to restrict the search space. If the user knows about a specific lawsuit in Alaska, performing an exemplar query would retrieve other similar lawsuits. The proposed methods offer many applications also for researchers, statisticians and scholars that are not fully aware of the characteristics of the data but want to find interesting patterns or general information about a topic.

Search engines

The techniques and the methods presented here are natural applications for search engines, in which they can improve the user experience by interactively refining a user query, or by proposing interesting related results (with exemplar queries). Since search engines are constantly improved to return better results, being able to adapt them to user needs and to propose alternative query paradigms is vital and, as such, one of the key goals of this thesis. Exemplar queries, for instance, can be employed as an additional service for people seeking information by providing examples or query recommendations [BYHM04].

Structured databases

Query reformulation in structured data (explored in Chapter 4) can serve as an amazingly handful service for the biology domain. Biological and chemical databases usually contain molecules or compounds that are interpreted by experts to find drugs or specific structures. Searching in such databases is problematic since most of the structures are repeated. As an exploratory tool, query reformulation as well as query relaxation, are used to suggest possible alternative queries that assist the expert in the analysis of the available data.

Mobile devices

Usability is, potentially, the core goal of mobile devices [BFJ+01, CSA07], such as ebook readers, mobile phones, and smart watches. Most of them guide the user towards the desired functionality through small steps, the so called *wizards*. The proposed interactive methods for empty/many-answers clearly respond with this trend, reducing the amount of information the user has to read from the screen. Visualizing a large set of results in small screens is troublesome, for it requires many swipe/scroll actions on the window. Interactive methods can therefore enabling search capabilities over large datasets with a minimum amount of actions.

1.4 Outline

This dissertation is organized as follows. Chapter 2 discusses the prior work in the area of empty answer, information overload and inexpressible queries. In Chapter 3 we describe our framework for the empty answer problem. We present the probabilistic model and how it serves different objective functions at the same time. We also introduce the algorithmic solutions and the experimental assessment in terms of performance and user satisfaction. Chapter 4 presents our study on query reformulation on graphs, introducing the context and our objective function that optimizes coverage and diversity at the same time. We describe a solution with quality guarantee and real-time performance and evaluate the technique in several real and synthetic datasets. In Chapter 5 we introduce exemplar queries, a novel query paradigm in which the user query is a representative of the user's expected results. We show an application of this paradigm to knowledge graphs and we evaluate the results both experimentally and through a user study. Chapter 6 concludes delineating the main contributions and describing the promising directions that follow this dissertation.

Chapter 2

State of the art

The state of art in the area of query reformulation for empty-answer is presented in Section 2.1. In Section 2.2 we survey the existing approaches for information overload and the differences with graph query reformulation presented in Chapter 4. Finally, Section 2.3 first introduces generic techniques for assisting the user in the query formulation phase, and then, looking at the neighborhood area of query answering in graphs, clarifies why exemplar queries discussed in Chapter 5 constitute a completely novel paradigm.

This chapter is based on the related works presented in our publications. More specifically, it expands on empty-answer framework [MMBR+13], graph query reformulation [MBG15], and exemplar queries [MLVP14a] related works.

2.1 Empty Answer

A considerable amount of research has focused on solving the empty-answer problem. The two main solutions for this problem are top-k results retrieval and query reformulation.

2.1.1 Top-k results

The most natural way to deal with the empty/many-answers problem is the *ranked-retrieval* approach, where the task is to propose a *ranking function* that assigns a score to all items (even those that do not exactly match the query conditions) in the

database. The ranking function captures the preferences of the user, and returns the top-k ranked items. This approach can be very effective when the user is relatively sophisticated and knows what she wants, because the ranking function can be directly provided by the user. However, in the case of a *novice user* who is unable to provide a good ranking function, there have been many efforts to develop suitable system-generated ranking functions, both by Information Retrieval (IR) [BYRN11, MPV13] and database [ACDG03, CDHW04, CDHW06] researchers.

The top-k approach proposed in [ACDG03] is based on the *idf* measure and adapted for relational databases. A relational database defines a set of attributes that represent a specific structure of an object, such as "Name" or "Address". These attributes have a set of admitted values called domain and the combinations of values constitute the tuples, e.g., name="John", Address="Main Road" is a tuple. The *idf* of value v in the domain of the attribute A_i of a database $\mathcal{D}$ is the logarithm of the number of tuples n in the database, divided by the number of tuples having value v in attribute A_i, denoted as $F_i(v)$. Intuitively, it gives higher scores to tuple matching less frequent attribute values. This approach can be generalized to numerical attributes subdividing the attribute into buckets of a fixed size. Given a query, the score of a tuple is given by the sum of the *idf* score on the attributes matching the query conditions.

The other top-k solutions [CDHW04, CDHW06] instead of using a deterministic score as *idf*, exploit probabilistic models. The method first computes a set of relevant tuples for a specific query. This set is estimated from a collection of past queries executed in the system. The score of a tuple is computed as the posterior probability of attributes not matching the query terms $\bar{M}$, given the set of relevant tuples R, in formula $\frac{P(\bar{M}|R)}{P(\bar{M}|M,\mathcal{D})}$, where $\bar{M}$ is the complement of M.

At the same time, it has also been recognized [BRWD$^+$08, HHI10] that the rank-retrieval based approach has an inherent limitation for novice users, namely it is not an interactive process, and if the user does not like the returned results, it is not easy to determine how the system-generated ranking function should be changed. In this context, approaches such as query reformulation are preferred alternatives.

2.1.2 Query reformulation

The alternative way to cope with the empty-answer problem is *query reformulation*. *Query reformulation* (also known as *query modification* or *query rewriting*) is a classic problem in databases and information retrieval, whose main objective is to provide the user with a set of alternative queries that better capture her search intent. Query reformulation is particularly useful in two complementary situations: (a) the user query is too specific and the result set is unavoidably empty (*empty answer*); (b) the query is too generic and the system returns a large number of results (*information overload*).

Query reformulation approaches are mostly non-interactive: they do not take into account the feedback of the user in the reformulation process. One of them [Cha90] proposes query modification based on a notion of generalization, and identifies the conditions under which a generalization is applicable. A generalization of a query Q on a database $\mathcal{D}$ is a query Q' such that $\mathcal{D}_Q \subseteq \mathcal{D}_{Q'}$, where $\mathcal{D}_Q$ is the set of results. A more recent work [KLTV06] suggests alternative queries on numerical attributes based on the "minimal" shift from the original (join) query. For a given empty-answer query Q, two operators are applied, namely RELAX and SKYLINE returning the shift between the numerical value and the constraints in the query and the tuple pairs that covers all the other tuples on the conditions of Q, respectively. Enabling reformulations of keyword queries on structured data is proposed in [YCHH12]. Queries are reformulated exploiting semantic (hierarchies) and structural (data schema) information in a relational database. The limitation of these approaches is that the user cannot change the preferences in the system. Our empty-answer framework, presented in Chapter 3, is an example of interactive approaches that can greatly help novice users in query answering.

Automatic query reformulation strategies for keyword queries over text data have been widely investigated in the IR literature [GS93, GS91]. To find related queries, various strategies have been proposed, including measures of query similarity [BYHM04], query clustering [WNZ02], or summary information contained in the query-flow graph [ABCG10]. An alternative approach relies on suggesting keyword relaxations by relaxing the ones which are least specific based on their *idf* score [HHI10]. Other notable approaches include the use of query-logs to retrieve search patterns [BBCV11] and learn reformulations. Relaxation strategies have been proposed as a recommendation service in [Jan07, JL06, Jun04]. Similar to our empty-answer framework, the solution proposed

by Jannach [Jan07] explores the lattice of the possible relaxations obtained relaxing one condition from the query at the time. From the lattice the method returns the maximal non-empty queries, i.e., the set of non-empty queries such that the addition of any single condition to them will lead to an empty-answer query. These methods are clearly non-interactive and suggest relaxations with the objective of reaching an interesting answer that has the minimum number of attributes relaxed. Moreover, in the absence of external sources, such as query-logs, these methods fail to accomplish the required goal of finding query reformulations.

An interactive method that refines a query to satisfy certain query cardinality constraints is introduced in [MK09]. The proposed techniques are designed to handle queries having range and equality predicates on numerical and categorical attributes. The input is a conjunctive query Q on a set of n constrained attributes $A_1, ..., A_n$, where a constraint is denoted as $A_i \diamond v$ with value v, and $\diamond \in \{>, <, \leq, \geq, =, \neq\}$, and the output is a set of refinements with $n-1$ constraints. This problem is solved finding minimal so called *extended queries*, that returns all the tuples satisfying at least $n-1$ constraints. For instance if the input query is $(A_1 \diamond_1 v_1 \wedge ... \wedge A_n \diamond_n v_n)$, an extended query is $(A_1 \diamond v_1 \wedge ... \wedge A_{n-1} \diamond_{n-1} v_{n-1}) \vee ... \vee (A_2 \diamond_2 v_2 \wedge ... \wedge A_n \diamond_n v_n)$.

Other interactive approaches include "Why-Not" queries studied in [CJ09, TC10], where, given a query Q and a set S of results expected by the user, an alternate query Q' is designed such that (a) is similar to Q for a certain notion of similarity, (b) returns the missing results S, and (c) the size of the results set should be minimal. "Why Not" queries are non-interactive, and the extension of these methods to deal with the empty answer problem is not trivial. Indeed, a "Why-not" query assumes the user to be aware of some desired tuples S in the database, whereas in our case, no such set S is required to the user. In addition, user preferences in "Why-not" questions are not explicitly modeled. The proposed relaxations are those that minimize the distance with the original query and include the set S of desired results with the minimum number of additional tuples. The approach we propose, instead, is unique in that it is built on top of a principled probabilistic optimization framework that takes into account user knowledge and preferences and integrates multiple objectives at once.

2.2 Information overload in graph search

Information overload problem requires some ad-hoc techniques, although complementary to the empty-answer problem. While most of the techniques presented in Section 2.1 still apply to information overload, a number of solutions exist for this specific problem.

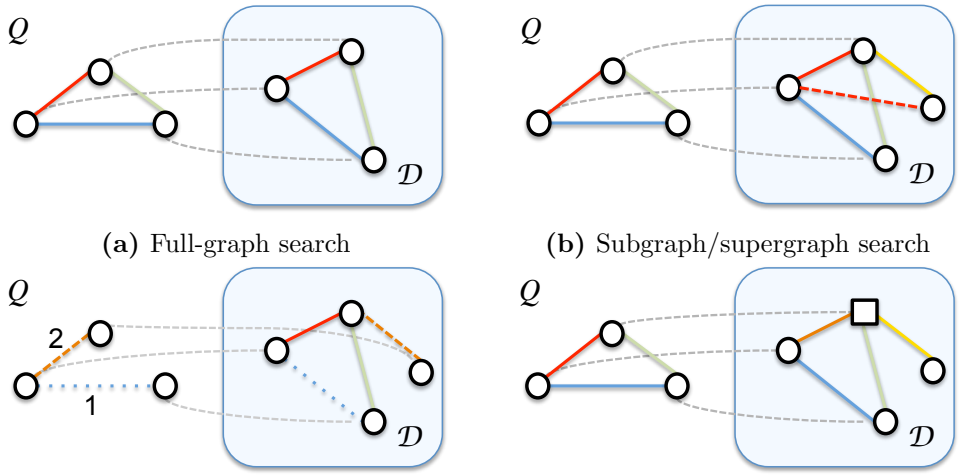
Returning the ranked top-k results is a way to prevent information overload. This method, highly used in information retrieval, conveys a number of limitations since the user can neither explicitly choose a subset of the results, nor can she see the rest of the results without navigating through them from top to bottom.

One effective method to overcome the information overload is *faceted search*. Faceted search refers to user interfaces that organize the results in a hierarchy of reformulated queries to enable the exploration of the query results. A recent study [BRWD$^+$08] proposes a minimum effort approach for faceted search, in which the query is refined adding one query condition at a time to drive the user towards the required results with the minimum number of interactions. This method tackles information overload using a decision-tree approach: given a query Q on a database $\mathcal{D}$ and a set of results $\mathcal{D}_Q$, it constructs a tree that minimizes the $Indg$ function that represents the number of indistinguishable tuples (tuples having the same value in a specific attribute). The $Indg$ is the score

$$Indg(A_i, \mathcal{D}_Q) = \sum_{v \in Dom_{A_i}} |\mathcal{D}_Q[A_i = v]|(|\mathcal{D}_Q[A_i = v]| - 1)/2,$$

with $\mathcal{D}_Q[A_i = v]$ being the set of tuples having value v in attribute A_i. Using the above equation the results are split across attribute values in a tree, such that the number of steps to find the desired answer is minimized. Similarly in spirit, Facetedpedia [LYR$^+$10] applies faceted search to Wikipedia.

As far as web search is concerned, query reformulation has also been used as a tool for a different problem, namely, *result diversification* [SMO10], that is the problem of selecting a subset of the query results that are diverse one another [DP12]. The latter work seeks the smallest set of points in a multidimensional plane that are r-DisC diverse. An r-DisC diverse set of points is such that any other point not in the set is at distance $\leq r$ to at least one point in the set. Intuitively, the set of r-DisC points are circles in the



(a) Full-graph search **(b)** Subgraph/supergraph search

(c) Generalized graph search **(d)** Similarity graph search

Figure 2.1: A query Q in a database $\mathcal{D}$ as a graph search.

plane that cover all the points with limited overlap between them. Although finding such a set is **NP**-hard, a greedy approximate algorithm exists. They also study zoom-in and zoom-out methods that allow real time adaptations of the parameter r. Diversity (along with coverage) is also at the basis of the solution presented in Chapter 4, even though with a different purpose: we want to find other queries that identify diverse subsets of the results of the original query rather than simply selecting a number of diverse query results.

All the aforementioned methods work with structured or textual data, however, to the best of our knowledge, no solution exists for graphs. We are particularly interested in labeled graphs, having (textual) descriptions in nodes and edges, for they can convey a vast amount of different information. This section first surveys the main related literature in answering graph queries, to then describe the neighborhood area of pattern discovery as a naïve approach for query reformulation in graphs presented in Chapter 4.

Querying graphs

A graph query is a structure (i.e., a graph) that needs to be found in the data graph. The types of queries studied in the literature include *full-graph search* [BDBV01], whose goal is to retrieve all graphs that are isomorphic to the input query and *subgraph/-supergraph search* [YYH05, CKNL07, SZLY08, CKN09, YM13], which finds all graphs that are supergraph/subgraph of the input query. A more flexible approach, the *generalized subgraph search* [LXCB12], is a generalization of subgraph search where exact edge matching is replaced with the notion of path matching constrained by a path

length. *Similarity search* [SLZ+10] relaxes the match conditions even further, returning all graphs that are similar enough to a query according to a given proximity measure. Figure 2.1 shows examples of the different graph search types in a small database. Given a query Q, full-graph search (Figure 2.1a) matches the query only if the matched structure is exactly the same as the query. Subgraph search, depicted in Figure 2.1b, finds substructures of the graph matching the query. In generalized graph search, the query specifies also a maximum distance at which a desired pattern should be found. In the example in Figure 2.1c the generalized query defines that a dashed edge must be found within two steps from a dotted edge. Figure 2.1d represents the most flexible method to match graphs, in which a similarity measure is provided to match results that resemble to some extent to the input query.

Graph pattern mining

Graph query reformulation reminisces the problem of finding patterns in graph databases. Research in this field has mainly focused on *frequent subgraph mining* [YH02, HWPY04, NK04], which aims at finding all structures (usually subgraphs, but also trees or paths) that occur frequently in the graphs of the database, and *optimal graph pattern mining*, that is the problem of finding substructure(s) that maximize a given quality function [MS00, YCHY08].

The earliest works in this area, AGM [IWM00] and FSG [KK01], propose Apriori-based algorithms. An initial empty pattern is expanded in a breadth first fashion exploiting the anti-monotonicity property of the frequency, for which the frequency of a pattern never exceeds the frequency of a subpattern. The exploitation of the Apriori property enables a fast pruning of the non-frequent patterns. Breadth first approaches have been demonstrated being slower and less performing than depth first search (DFS) approaches, like GSpan [YH02]. GSpan is the first work proposing the concept of DFS codes, a kind of canonical code that uniquely represents a graph pattern as a string. Therefore, equal graph structures implies equal canonical codes and vice versa. On this basis, a number of approaches have been proposed elaborating over the concept of informative patterns [KCN07, SMG09]. Intuitively, a pattern is informative if it can be used as a description of a set of similar patterns. In [SMG09] the notion of informativeness is captured by the correlation. Given two graphs G_1, G_2, the correlation is $\frac{Cov(G_1,G_2)}{\sigma(G_1)\cdot\sigma(G_2)}$, where Cov is the covariance and σ the standard deviation. If a graph correlates with one of its subgraphs we conclude that it does not convey more information and thus can be

ignored. Compact indexes of patterns can be constructed discarding highly correlated patterns.

Frequent subgraph mining might be viewed as a naïve approach to query reformulation in graph databases: find the supergraphs of the query graph that appear frequently in the database and just interpret them as reformulations. The notion of informativeness is somehow related to the diversity proposed in our approach. However, the aforementioned methods ignore the user query that is instead the starting point in our approach. This issue is expanded in Chapter 4.

Result-set reduction

The problem of reducing the answers to a query issued to a graph database has also received some attention in the literature. Existing solutions rely on either clustering the graphs in the result set [FVS+09, HCY+12] (not to be confused with the problem of clustering the vertices of a *single* graph [Sch07]) or returning top-k representative results [RHS14]. A top-k representative query is a query on a graph database (remember that a graph database is a set of small graphs) that returns the k representative graphs that compactly summarize all other results. Each graph g in the database $\mathcal{D}$ is a feature vector $\vec{g} = [g_i, ..., g_m]$, in which a feature is a graph pattern and the i-th value of the vector is 1 if and only if g_i is a subgraph of g. A top-k representative query returns the subset of the results matching the query, i.e., $\{g \in \mathcal{D} | Q(\vec{g}) = 1\}$, that maximizes a *budget function*. The budget function measures the number of results that are similar to those in the considered subset. Returning the top-k representative queries follows the trend of overcoming the information-overload issue, but is only tangentially related to the problem we tackle in Chapter 4. Our query-reformulation problem is indeed different: rather than aiming at reducing the result set, we ask for something more, i.e., we want to output a set of reformulated (i.e., more specific) queries that can help the user better comprehend the results and refine her search. Existing approaches to result-set reduction cannot instead output query reformulations, as our problem requires.

2.3 Inexpressible queries

Sometimes, novice users have to cope with the problem of the limited expressive power and usability of the database systems. Usability has been a key issue in most of the modern database systems for the last decades [Dat83, JCE+07]. Although a single

definition does not exist, usability is intuitively defined as the degree to which users can access, learn, and enjoy a particular system with their own skills [Car99].

2.3.1 Database usability

One of the earliest notable works addressing usability in relational databases is *query-by-example* (QBE) [Zlo75], a methodology, orthogonal to SQL, to query by filling missing values in tuple-centric interfaces. Similarly, more recent works have been successfully proposed for XML data, such as XQBE [BCC05]. This is only marginally related with the spirit of this dissertation, in that their main focus is on alternative query interfaces, while we aim to assist the users (especially novice users) in the query answering process and hide most of the details of the schema and the structure of the data.

Visual interfaces for querying have a long history. Apart from successful industrial applications, such as Microsoft SQL Server Data Tools [sql] and Oracle Developer Tools [ora], several approaches have been proposed in this regards, such as GRIDS [SY98] and FoXQ [Abr03] to name a few. Works in this area are meant to simplify the access to the data allowing for an interactive design of the query. Along the same line, CURSED [PPV02] proposes a method to easily create forms. This visual approach is somehow orthogonal to the one presented in this work. We intend to improve the query capabilities of the system and, instead of facilitating the user in the query creation part, we offer novel solutions in case of failure of the query answering process.

Usability drives the study of keyword-based interfaces, in which the user describes the desired answer in natural language. These interfaces are inherently more intuitive and suitable for novice users. However, they convey a number of issues related to the ambiguity of natural language. Solutions have been proposed to answer keyword queries over relational [BDG$^+$11, BGS14], trees [FKM00, GSBS03], and graphs [PHIW12, KA11, KRS$^+$09]. The reliability of these methods depends on NLP (Natural Language Processing) techniques and ranking models. Our approach, instead, is more database-centric, in that the ultimate goal is to enrich the semantics of the query answering systems. Moreover, most of the visual interfaces assume that the user is familiar with the relational model and can formulate the query in the correct way. The latter assumption, in particular, does not hold for novice users.

2.3.2 Query answering in knowledge graphs

We study the problem of Inexpressible queries when the data model is a graph, as we did for the many-answer problem. More specifically, most of the data is conveniently represented as a *knowledge graph*. Knowledge graphs are labeled graphs whose nodes represent entities (e.g. persons, locations, concepts) and edges are relationships between them. Typical examples of knowledge graphs include Freebase [BEP+08], YAGO [SKW07], and DBpedia [LIJ+15]. This model accommodates a plethora of needs and enables novel interesting challenges [LMP+14].

Related literature for Inexpressible queries includes query modification, as already introduced in the previous section. However, if the user does not know at all how to formulate the query conditions, the aforementioned techniques cannot be used. As such, in this specific case, query reformulation is not a valid alternative.

Query answering in knowledge graphs is mainly achieved with *keyword search* and *approximate search*.

Graph Keyword search

Keyword search is also a convenient way to cope with Inexpressible queries. The user provides a keyword query $Q = (w_1, ..., w_s)$, that consists of a list of words w_i, on the knowledge graph $\mathcal{D}$, and receives as answers a set of graphs or matched nodes. Table 2.1 presents the main works in this area divided by the shape of the structures matched in the knowledge graph.

Tree-based methods [HWYY07, KRS+09] return a tree as an answer to a keyword query. BLINKS [HWYY07] is one of the first works tackling keyword search in graphs efficiently. This method exploits fast indexing techniques to retrieve trees connecting the set of matched nodes. The answers are then ranked using a scoring function that integrates graph measures (e.g., node degree) and IR measures (e.g., tf-idf). The more recent STAR [KRS+09] extends the idea of returning trees as answers to steiner trees [1]. Since the general Steiner tree problem is **NP**-hard, they propose efficient approximations with quality guarantees.

A different approach is the one taken by [KEW09] and [KA11]. MING [KEW09] extends STAR returning graphs as results. The relevance of a result is computed with a random

[1] A steiner tree is a tree having the minimal distance between a set of input nodes.

Output shape	Description	Previous Work
Trees	These methods produce connected trees as output, either steiner trees or normal trees	[HWYY07], [KRS+09]
Subgraphs	These methods return a connected subgraph of the knowledge base.	[KEW09],[KA11]
Structured queries	These methods match seed nodes with any structure. The returned output can be disconnected.	[PHIW12]

Table 2.1: Keyword search on knowledge graphs literature

walk on the graph starting by the candidate matching nodes. Similarly, r-clique [KA11] envisioned the advantages of returning subgraphs that minimizes the overall distance between the matched nodes, materializing in an index all the possible paths of size at most r. A valid r-clique must include a node for each candidate match of the query terms.

Logic-based approaches look at the problem from a different angle. In this sub-area, QUICK [PHIW12] proposes a hybrid method that exploits NLP parsing to annotate query terms as entities, concepts and relationships, and subsequently builds a structured query that resembles a description logic predicate. This predicates are then evaluated on the graph and ranked according graph and IR measures.

Approximate search in graphs

An alternate approach is approximate search. Approximate search can be performed when the user does not know the exact conditions to formulate the query and sends an incomplete or imprecise query. On graphs, p-homomorphism [FLM+10] enables similarity structure search instead of the strict isomorphism. Likewise, NeMa [KWAY13] introduces the notion of node neighborhood (i.e., the set of nodes reachable from a source node in a fixed number of steps) to match nodes and edges within some error range, introduced by the user in the query. One recent work, SLQ [YWSY14], expands the latter technique including a ranking model for a set of fixed textual/topological transformations from query nodes to answer nodes in the graph. Noticeably, more elastic notions of graph match have been proposed exploiting *strong simulation* to answer a query. Strong simulation is a recently introduced form of pattern search [MCF+14] that considers the query as a transition process (where nodes are states of the system and edges are transitions between the states), and finds answers that preserves the same transitions and the same order of the user query. Interestingly, the latter problem admits a polynomial

time algorithm. Simulation is also explored in one our recent work [MLVP15]. An approach close to ours is GQBE (Graph Query by Example) [JKL$^+$13, JKL$^+$14], where an input set of nodes is treated as an initial example seed for the results of the user. Approximate search requires at least an incomplete description of the query, falling short to fulfill the goal of the inexpressible queries problem, that is instead addressed by our work in Chapter 5.

Chapter 3

A Holistic Approach for the Empty-Answer Problem

We new focus on the *empty answer* problem and, more specifically, on the study of interactive methods. The traditional approach for the *empty answer* problem is query relaxation that proposes relaxed queries with less constraints than the user query. Interactive methods are more suitable for novice users in that they are designed to ask "yes/no" questions instead of showing a set of relaxations all at once. Nevertheless, when dealing with many stakeholders, such as sellers and end-users, the system should adapt and return answers based on different objectives and preferences. In this chapter, we present a principled optimization framework for the empty-answer problem, that accepts a wide range of objectives. The framework embeds a probabilistic model of the user preferences and beliefs. We also include experimental results and a user study to demonstrate the quality of the results of our framework. The algorithms have also been presented in a system demonstration [MMBR$^+$14].

3.1 Contributions

The main contributions in this chapter can be summarized as follows.

- We propose a principled probabilistic optimization-based interactive framework for the empty-answer problem that accepts a wide range of optimization objectives,

and is based on estimation of the user's prior knowledge and preferences over the data.

- We propose novel algorithmic solutions using our framework. The algorithms FastOpt and CDR produce optimal and approximate relaxation sequences respectively, without having to explore the entire relaxation tree.
- We propose an extension of the framework that returns top-k relaxations at each step, and also we allow the possibility to specify a cardinality constraint on the size of the results. Enabling top-k relaxations is a critical step that affects time. As such, we introduce a new algorithm, FastCDR, that embeds both the optimal FastOpt pruning and the approximate cost computed by CDR.
- We explain how our techniques can be used for different cases such as categorical attributes with hierarchies, numerical attributes, or cases with constraints on the expected answer set.
- We perform a thorough experimental performance and scalability evaluation using different optimization objectives on real datasets, as well as a usability study on real users, and we report our findings.

3.2 Outline

The chapter is organized as follows. In Section 3.3 we present a motivating example used through the rest of the chapter. Section 3.4 introduces the probabilistic framework and the notation used to define the exact and approximate algorithms in Section 3.5. Extensions to the framework are described in Section 3.6. Finally, in Section 3.7 we present experimental results on real data as well as usability studies on real users.

3.3 Motivating example

Consider a web site like *cars.com*, where users can search for cars by specifying in a web-form the desired characteristics. An example instance of such a database is shown in Figure 3.1. A user is interested in a car that has anti-lock braking system (ABS), dusk-sensing light (DSL), and manual transmission. The data instance of Figure 3.1 reveals that there is no car that satisfies these three requirements.

	Make	Model	Price	ABS	MP3	Alarm	4WD	DSL	Manual	HiFi	ESP	Turbo
t_1	VW	Touareg	$62K	1	0	0	0	0	1	0	1	0
t_2	Askari	A10	$206K	0	1	0	0	1	1	1	1	0
t_3	Honda	Civic	$32K	1	0	0	0	0	0	0	0	0
t_4	Porsche	911	$126K	0	0	0	0	1	0	1	1	0

Figure 3.1: An instance of a car database.

The user is in an urgent need of a cheap car, and is therefore willing to accept one that is missing some of the desired characteristics. The system knows that the cheapest car is a Honda Civic (i.e., tuple t_3) that has ABS, but no manual transmission and no DSL. So it proposes to the user to consider cars with no Manual transmission. If the user accepts, the system next propose to the user to consider cars with no DSL. If she also accepts the second relaxation, then the cheapest car of the database, tuple t_3 would be returned.

Instead, the system could also propose to the user cars with no ABS in the beginning. However, if the user accepts that suggestion, this would result in a match of the most expensive car of the database (Askari A10, tuple t_2). Since the user wishes to find the cheapest car, therefore, proposing first to relax the DSL requirement is preferable.

Assume that when the user is first asked to relax DSL, the answer is no. In this case, the system needs to investigate what alternative relaxations are acceptable. If the system knew that most users prefer cars with DSL, it could have used this knowledge to propose a different relaxation in the first place. In the following sections, we present a framework that takes into account all the above issues, for different optimization objectives.

The set of possible relaxations of the query $ABS{=}1 \wedge DSL{=}1 \wedge Manual{=}1$ is graphically depicted in Figure 3.2 as a lattice where each node represents a query. The query of a node is a relaxation of the query modeled on the node above. The query is expressed as a triple where each value of the triple means that the respective condition is ignored if "-" or considered if "1" (e.g., $ABS{=}1 \wedge DSL{=}1$ as $(1, 1, -)$). The original query can be modeled as $(1, 1, 1)$, depicted at the root of the lattice, while each of the other nodes in the lattice represents a relaxed query. A directed edge from node p to node p' denotes that p' contains exactly one additional relaxation that is not present in p. For

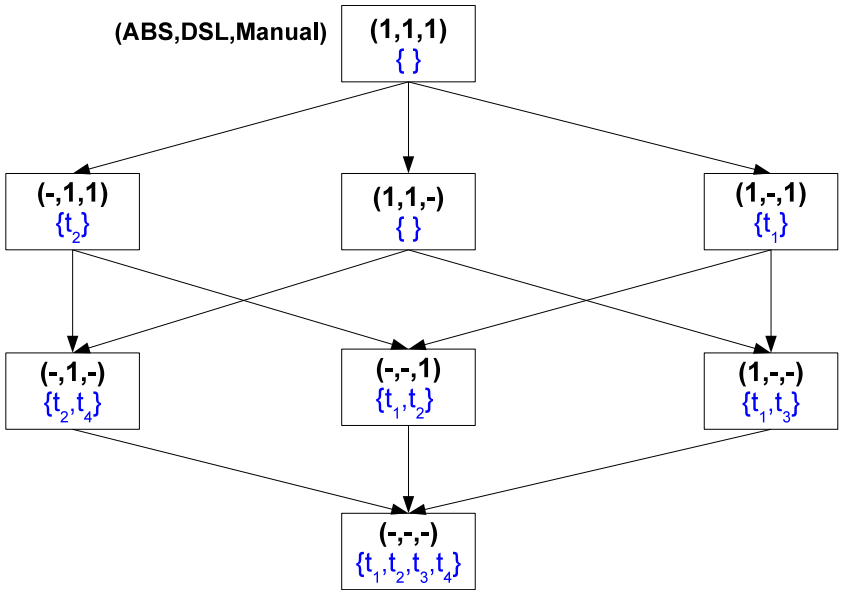


Figure 3.2: Query lattice of the query Q in Example 3.1.

illustration, each relaxation contains the tuples in its answer set. Note that, given a query with k conditions, the number of possible relaxations is exponential in k.

3.4 Background and Problem

This section introduces the proposed probabilistic framework for interactive query relaxation. The framework is based on a cost associated to each user interaction with the system. Given the generality of the cost model, application-specific instantiations are presented (such as, proposing results with the maximum price). For ease of explanation we consider databases with only Boolean attributes and we defer to Section 3.6 for a discussion on how to convert any database to Boolean.

3.4.1 Background

Let $\mathcal{A}$ be a collection $\{A_1, A_2, \ldots, A_m\}$ of m attributes, with each attribute $A_i \in \mathcal{A}$ associated with a finite domain Dom_{A_i}. The set of all possible tuples $\mathcal{U} = Dom_{A_1} \times Dom_{A_2} \times \ldots \times Dom_{A_m}$ constitutes the *universe*. A *database* is a finite set $\mathcal{D} \subseteq \mathcal{U}$. A tuple t $(v_1, v_2, \ldots, v_m)$ can also be expressed as a conjunction of conditions $A_i = v_i$, for $i = 1..m$, allowing it to be used in conjunctive expressions without introducing new operators. Given $t \in \mathcal{U}$ we denote as $\mathfrak{Constrs}(t)$ the set of conditions of t.

A *query* Q is a conjunction of *atomic* conditions of the form $A_i{=}v_i$, where $A_i{\in}\mathcal{A}$ and $v_i{\in}Dom_{A_i}$. Each condition in the query is referred to as *constraint*. The set of constraints of a query Q is denoted as $\mathfrak{Constrs}(Q)$. A query can be equivalently represented as a tuple $(v_1, v_2, \ldots, v_m)$ where the value v_k corresponds to attribute A_k and models the condition $A_k{=}v_k$ if $v_k{\in}Dom_{A_k}$ or the boolean value "true" if v_k has the special value "-". Similarly, a tuple $(v_1, v_2, \ldots, v_m)$ can be represented as a query Q, i.e., a conjunction of conditions of the form $A_i{=}v_i$, one for each value v_i. Thus, by abuse of notation, we may write a tuple in the place of a query. A tuple t *satisfies* a query Q if $\mathfrak{Constrs}(Q) - \mathfrak{Constrs}(t){=}\emptyset$. The *universe* of a query Q, denoted as $\mathcal{U}_Q$, is the set of all the tuples in the universe $\mathcal{U}$ that satisfy the query Q. The answer set of a query Q on a database $\mathcal{D}$, denoted as $\mathcal{D}_Q$, is the set of tuples in $\mathcal{D}$ that satisfy Q. It is clear from the definition that $\mathcal{D}_Q \subseteq \mathcal{U}_Q$. An empty answer to a user query means that none of its satisfying tuples are present in the database.

Example 3.1. *The tuple* t_1 *in Figure 3.1 can be represented as* $Make{=}VW \wedge Model{=}Touareg \wedge Price{=}62K \wedge ABS{=}1 \wedge Computer{=}0 \wedge Alarm{=}0 \wedge 4WD{=}0 \wedge DSL{=}0 \wedge Manual{=}1 \wedge HiFi{=}0 \wedge ESP{=}1 \wedge Turbo{=}1$. *Given the set of attributes (ABS, DSL, Manual), the query* $ABS{=}1 \wedge DSL{=}1 \wedge Manual{=}1$ *can be modeled as* $(1, 1, 1)$ *while the query* $ABS{=}1 \wedge DSL{=}1$ *as* $(1, 1, -)$. ■

3.4.2 Generic Probabilistic Framework

A relaxation is the omission of some of the conditions of the query. This results into a larger query universe, which means higher likelihood that the database will contain one or more of the tuples in it, i.e., the evaluation of the relaxed query will return a non-empty answer.

Definition 3.1. A *relaxation* of a query Q is a query Q' for which $\mathfrak{Constrs}(Q'){\subseteq}\mathfrak{Constrs}(Q)$. The constraints in $\mathfrak{Constrs}(Q) - \mathfrak{Constrs}(Q')$ are referred to as *relaxed constraints* and their respective attributes as *relaxed attributes*. ■

For the rest of this chapter, since our goal is to provide a systematic way of finding a non-empty answer relaxation, we consider for simplicity only relaxations that involve one constraint at a time. Note that there are other forms of relaxations, relevant to categorical attributes with hierarchies, or numerical values. Our techniques can also handle these forms of relaxations. We discuss these cases further in Section 3.6.

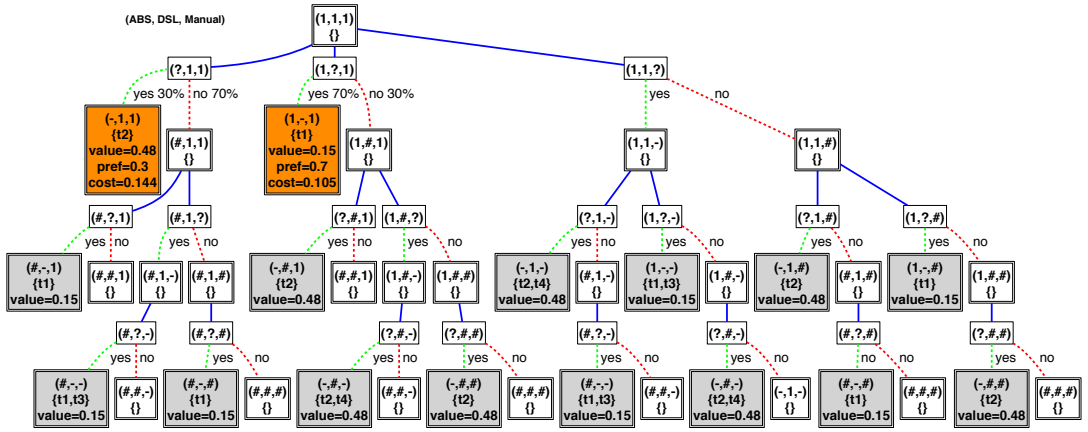


Figure 3.3: Query Relaxation tree of the query in Example 3.1.

The extra tuples that the query universe of a relaxation of a query Q has as opposed to query universe of Q is called a *tuple space*.

Definition 3.2. The *tuple space* of a relaxation Q' of a query Q, denoted as $\mathbb{TS}_Q(Q')$, is the set $\mathcal{U}_{Q'} - \mathcal{U}_Q$. ■

Among the constraints of a user query, some may be fundamental and the user may not be willing to relax them. We refer to such constraints as *hard constraints* and to all the others as *soft constraints*. Since the hard constraints cannot be relaxed, for the rest of this chapter we focus our attention on the remaining constraints of the user query, which are initially considered to be soft.

In the tuple representation of a query, we use the "#" symbol to indicate a hard constraint, and "?" to indicate a question to the user for the relaxation of the respective constraint.

Example 3.2. *The expression* $(1, \#, -, 1, ?)$ *represents a relaxation query for which the user has already refused to relax the second constraint (i.e., she has kept the original query condition on the second attribute as is), has accepted to relax the third one, and is now being proposed to relax the last constraint.*

In order to quantify the likelihood that a possible relaxation Q' of a query Q is accepted by the user, we need to consider two factors: first, the *prior* belief of the user that an answer will be found in the database using the relaxed query Q', and second, the likelihood that the user will *prefer* (i.e., be satisfied with) the answer set of Q'. The relaxation

Q' selected by the framework should have high values for both factors, and additionally should attempt to optimize application-specific objectives (e.g., try to steer the user towards highly profitable/expensive cars). We provide generic functional definitions of both factors next, and defer application-specific details to Section 3.4.4.

Since we cannot assume that the user knows any tuple in the database, we resort to a probabilistic method for modeling that knowledge through a function called $prior(t, Q, Q')$. It specifically measures the *user belief* that a certain tuple t satisfying the relaxed query Q', i.e., a tuple from the tuple space of the relaxation, exists in the database. In order to estimate the likelihood that the user is satisfied with an answer set, we use a preference function $pref(t, Q)$ that captures the probability that a user will like a tuple t, given the query. Section 3.4.4 discusses how specific prior and pref functions can be constructed for various applications.

Using the prior and the pref functions, we can compute the *relaxation preference function*, i.e., the probability that a user accepts a proposed relaxation Q' to a query Q (where Q evaluates to an empty answer). The probability to reject the relaxation is:

$$relPref_{no}(Q, Q') = \sum_{t \in \mathbb{TS}_Q(Q')} (1 - pref(t, Q')) * prior(t, Q, Q') \tag{3.1}$$

which represents the probability of not liking any of the tuples in the tuple space. Thus the probability of accepting the relaxation is the probability that the user likes at least one tuple, which is the inverse of the probability of the user not liking any tuple (i.e., rejecting the relaxation), namely

$$relPref_{yes}(Q, Q') = 1 - relPref_{no}(Q, Q') \tag{3.2}$$

To encode the different relaxation suggestions and user choices that may occur for a given query Q that returns no results, we employ a special tree structure which we call the *query relaxation tree* (see Figure 3.3 for an example of such a tree). This is similar to tree structures used in machine learning techniques and games [Mit97]. The tree contains two types of nodes: the *relaxation* nodes (marked with double-line rectangles in Figure 3.3) and the *choice* nodes (marked with single-line rectangles in Figure 3.3).

Note that the children of relaxation nodes are choice nodes, and the children of choice nodes are relaxation nodes.

A *relaxation node* represents a relaxed query. The root node is a special case of a relaxation node that represents the original user query. A relaxation node does not have any children when the respective query returns a non-empty answer, or returns an empty-answer but cannot be relaxed further (either because all its constraints are hard, or because no further relaxation can lead to a non-empty answer). In every other case, relaxation nodes have k children, where k is the number of soft constraints in the query corresponding to the node. Each child represents an attempt to further relax the query. In particular, the i-th child represents the attempt to relax the i-th soft constraint (recall that in each interaction step we attempt to relax only a single constraint). These children are the choice nodes.

A *choice node* models an interaction with the user, during which the user is asked whether she agrees with the relaxation of the respective constraint. Each choice node has always two children: one that corresponds to a positive response from the user, and one that corresponds to a negative response. In the first case, the child is a relaxation node that inherits the constraints from its grandparent (i.e., the closest relaxation node ancestor), minus the constraint that was just relaxed (this constraint is removed). In the second case, the child is a relaxation node inheriting the constraints from the same grandparent, but now the constraint proposed to be relaxed has become a hard constraint (the relaxation was rejected). A choice node can never be a leaf. Thus, any root-to-leaf path in the tree starts with a relaxation node, ends with a relaxation node, and consists of an alternating sequence of relaxation and choice nodes.

Each path of the tree from the root to a leaf describes a possible relaxation sequence. Note that if the query Q consists of k constraints (i.e., attributes), there are an exponential (in k) number of possible relaxation sequences. In practice, the number of paths is significantly smaller, because they may terminate early: at relaxation nodes that have a non-empty answer, or at relaxation nodes for which no descendant corresponds to a non-empty answer.

Example 3.3. *Figure 3.3 illustrates the query relaxation tree for the query Q in the Example 3.1. Relaxation nodes are modeled with a double-line and choice nodes with a single-line border. The color-filled nodes are nodes corresponding to relaxations with a*

non-empty answer. The non-colored leaves correspond to relaxations that cannot lead to a non-empty answer, irrespective of further relaxations that may be applied. Notice how the "?" symbol is used to illustrate the proposal to relax the respective condition, and how this proposal is turned into a relaxed or a hard constraint, depending on the answer provided by the user. ■

Next, we introduce and assign a *cost* value to every node of the query relaxation tree. Having the entire query relaxation tree that describes all the possible relaxation sequences, the idea is to consider the cost value of each relaxation node to determine which relaxation to propose during each interaction, based on the specific optimization objective, as we describe in Section 3.4.4.

Recall Equations (3.1) and (3.2) that describes the probability that a user will reject, or accept a specific relaxation proposal made by the system. Using these formulae, in general, the cost of a choice node n can be expressed as:

$$\begin{aligned} Cost(n) = & relPref_{yes}(Q, Q') * (C_1 + Cost(n_{yes})) + \\ & relPref_{no}(Q, Q') * (C_1 + Cost(n_{no})) \end{aligned} \tag{3.3}$$

where the n_{yes} and n_{no} are the two children (relaxation) nodes of n, Q is the query corresponding to the parent of n, and Q' corresponds to the suggested relaxation of Q at node n. In the formula, the variable C_1 is a constant, that is used to quantify any additional cost incurred for answering the current relaxation proposal.

The cost of a relaxation node, on the other hand, depends on the way the costs of its children are combined in order to decide the next relaxation proposal. To produce the optimal solution, at every step of the process, a decision needs to be made on what branch to follow from that point onward. This decision should be based on the selection of the relaxation that optimizes (maximizes or minimizes) the cost. Thus, the cost of a relaxation node n in the query relaxation tree is

$$Cost(n) = optimize_{c \in S} Cost(n_c) \tag{3.4}$$

where Q is the query that the node n represents, S is the set of soft constraints in $\mathfrak{Constrs}(Q)$, and n_c is the choice child node of n that corresponds to an attempt to relax

the soft constraint c. The optimization task is either maximization or minimization depending on the specific objective function.

The cost of a leaf node depends on a specific optimization objective, and the "value" of the tuples in that leaf node in contributing towards this objective. These details are presented in Section 3.4.4.

Therefore, the final *task* is to propose a sequence of relaxation suggestions interactively, such that the cost of the *root* node in the relaxation tree is optimized. An algorithm for that task using the relaxation tree is discussed in Section 3.5.1.

3.4.3 Theoretical analysis

Consider the relaxation problem for an empty answer query in which the aim is to minimise the user effort, i.e., the number of user interactions needed. If one is able to find that minimum cost relaxation, the simpler problem of deciding whether a relaxation has a cost at most n, should have the same or smaller complexity. Unfortunately, the latter problem can be shown to be **NP**-complete.

Theorem 3.3. *Given a database $\mathcal{D}$ and an empty answer query Q, deciding whether there is a query relaxation tree such that the cost of its root node is less than or equal to a constant n is* **NP***-complete.*

Proof. For the proof we can assume that we have a boolean database, i.e., a database where each attribute takes a boolean value of 0 or 1. We will show that even for that special case, the problem is still **NP**-complete. To do so we reduce the known to be **NP**-complete exact cover by 3-set problem (denoted as X3C) [MD79] to ours. Given a finite set U over $3n$ elements, and a collection S of 3-elements subsets of U, X3C finds an exact cover for U, i.e., a sub-collection $C \subseteq S$ of subsets, such that every element of U occurs in exactly one member of C.

Consider an instance X3C(U, S) of the X3C problem that consists of a finite set $U = \{x_1, x_2, \ldots, x_{3n}\}$ defined over $3n$ elements, and a family $S = \{S_1, S_2, \ldots, S_q\}$ of subsets of U, such that, $|S_i| = 3, \forall 1 \leq i \leq q$ and requires a yes or no answer on whether there exists a cover $C \subseteq S$ of n pairwise disjoint sets, covering all elements in U.

Given a specific instance $\mathcal{I}$ of the X3C problem, we create a database D with q boolean attributes $A = \{A_1, \ldots, A_q\}$ and $3n$ tuples $T = \{t_1, \ldots, t_{3n}\}$. For each $S_i = \{x_j, x_k, x_l\}$, A_i contains boolean 1 for tuples $\{t_j, t_k, t_l\}$, while the remaining tuples get boolean value 0 for attribute A_i. This way, every attribute is present (i.e.,corresponds to 1 value for that attribute) in only three tuples. For such a database, we consider the query $Q \in \{0\}^q$ (i.e., all q constraints of Q are set to 0). We also construct a simple "black-box" ranking function that assigns a preference score $pref(t, Q)$ to each tuple t in the database. $pref(t, Q) = 1$, when the tuple exactly matches all the query predicates; otherwise, $pref(t, Q) = 0$. Therefore, for our instance, it is easy to see that $pref(t, Q) = 0$ for all the tuples, since Q returns no answer in the first place. The next step is to generate a relaxation tree, and compute the cost of its root node based on the minimum cost strategy. Interestingly, using the black-box preference function described above, we have $relPref_{no} = 100\%$ and $relPref_{yes} = 0$ in each response node. This is indeed true, because, $pref(t, Q) = 0$ for every tuple t, which results in $relPref_{yes} = 0$ for every response node. The above steps achieved to create an instance $\mathcal{J}$ of the query relaxation problem from an instance I of the X3C problem that we initially considered.

We claim that $\mathcal{I}$ is a YES-instance of X3C iff $\mathcal{J}$ is a YES-instance of our problem.

($\Rightarrow$): Suppose $C = \{S_{j_1}, ..., S_{j_n}\} \subseteq S$ is an exact (disjoint) cover of U. Then consider each node in the optimal path of the query relaxation tree, where tuple $t_i \in A_k$ iff $x_i \in S_{j_k}$. Notice that each node A_k consists of only three tuples. Since C is an exact cover of U, each element $x \in U$ appears in exactly one subset $S_k \in C$. Thus, $Cost(root) = 100\% * (1 + Cost(n_{no})) = n$, that indicates that $\mathcal{J}$ is a YES-instance.

($\Leftarrow$): Let π be a the optimal path in the query relaxation tree. $Cost(root) = n$, witnesses the fact that $\mathcal{J}$ is a YES-instance. Observe that any node in π only contributes to 3 tuples to the database. Since the overall cost is n, it follows that every node contains exactly 3 tuples and they are disjoint. Now, if we consider the collection S, it is easy to verify that $|C| = n$ and that every element $x \in U$ appears in exactly one set $S \in C$. Therefore, $\mathcal{I}$ is a YES-instance.

□

3.4.4 Application-Specific Instantiations of the Probabilistic Framework

The generic query relaxation framework presented in the previous section is largely agnostic to application-specific details. However, to illustrate its range of applicability, we take the opportunity here to discuss various specific instances of the framework, notably different instances of the *prior*, *pref*, and *objective functions*.

Recall that the *prior* function represents the user's prior knowledge of the content of the database. An implementation of the *prior* is to consider the data distribution in the case of known data domains. One possible implementation, which is the one we use, is the popular Iterative Proportional Fitting (IPF) [BFH07, PKM05] technique on the instance data (which can be thought as a sample of the subject domain) to estimate the required probabilities. IPF takes into account multi-way correlations among attributes, and can produce more accurate estimates than a model that assumes independence across attributes. However, we note that the independence model, or any other probability density estimation technique can be applied in the place of IPF.

The *pref* function is the probability/likelihood that a user will like a tuple t given a query. In simple instances, e.g., where the user in interested in cheap items in the query instances, the preference for a tuple can be modeled as any suitable function where the probability is dependent on the price of the item (higher the price, lower the probability). More generally, the approach is to use a tuple scoring function for calculating the *pref* of the tuples that imposes a non-uniform bias over the tuples in the tuple space. For example, instead of simple tuple scoring functions (such as price), one could also use more complex scoring functions such as assigning a *relevance* score [BYRN11] to each of the tuples. There exists a large volume of literature on such ranking/scoring functions [ACDG03, CDHW04, BYRN11]. Even though any of these functions are possible, in our implementation, we use a simple and intuitive measure, which is based on the Normalized Inverse Document Frequency [ACDG03].

$$pref(t, query) = \frac{\sum_{c \in \mathfrak{Constr}(query) \cap \mathfrak{Constr}(t)} idf(c)}{\sum_{c \in \mathfrak{Constr}(query)} idf(c)},$$
$$\text{where } idf(c) = \log \frac{|\mathcal{D}|}{|\{t | t \in \mathcal{D}, t \text{ satisfies } c\}|}.$$

However, the question remains - as the relaxation process progresses, does the preference of the user also evolve, i.e., the preference for a particular tuple changes? Note that the preference for a particular tuple may be computed in several different ways: (1) preference for a tuple is independent of the query and is always static - an example is where the preference is tied to a static property of the tuple, such as price, (2) preference for a tuple is query dependent, but only depends on the initial query and does not change during the interactive query relaxation session - e.g., when the preference is based on relevance score measured from the initial query, and (3) preference for a tuple is dependent on the latest relaxed query the user has accepted - this is a very dynamic scenario where after each step of the interactive session the preference can change. These different preference computation approaches are referred to as Static, Semi-Dynamic, and Dynamic respectively.

The generic probabilistic framework discussed in the previous subsection could be used to optimize a variety of *objective functions*, by appropriately modifying the preference computation approach of the tuples, and the cost computation of the leaf nodes, relaxation nodes, and the choice nodes of the relaxation tree. We illustrate this next.

Just as each tuple has a preference of being liked by a user, each tuple can also be associated with a *value* that represents its contribution towards a specific objective function. It is important to distinguish the value of a tuple from the preference for a tuple - e.g., if the objective is to steer the user towards overpriced, highly-profitable items, then the value of a tuple may be its price (higher the better), whereas the user may actually prefer lower priced items (lower is better) - although in most applications the value and the preference of a tuple are directly correlated. Thus in some applications, our query relaxation algorithms have to delicately balance the conflicting requirement of trying to suggest relaxations that will lead to high-valued tuples, but at the same time ensuring that the user is likely to prefer the proposed relaxation. The following example illustrates this situation:

Example 3.4. *Consider the example database in Figure 3.1, and assume that instead of steering the user towards cheap cars, the objective may steer the users towards expensive cars. In this case, the value/preference of a tuple is directly/inversely correlated with its price. For the purpose of illustration, let value of* $t_1 = 0.15, t_2 = 0.48, t_3 = 0.07, t_4 = 0.30$. *Let us also assume that the probability that the user will say "yes" to relaxing ABS is only* 0.3 *(e.g., she knows that most cars come with ABS systems, and relaxing ABS*

will not offer too many additional choice of cars), whereas the probability that she will say "yes" to relaxing DSL is much higher at 0.7 *(e.g., it may be a relatively rare feature, and relaxing it may offer new choices). Of course, our system can only estimate these relaxation preference probabilities using Equations 1 and 2, which depend on the prior and tuple preference functions.*

Then, the cost of relaxing ABS is the expected value that can be achieved from it, which is $0.3 \times 0.48 = 0.144$*, while the cost of relaxing DSL is* $0.7 \times 0.15 = 0.105$*. The system would therefore prefer to suggest relaxing DSL to the user, since it has a higher cost (i.e., potential for greater benefit towards to overall objective), even though* t_2 *has lower preference than* t_1*.* ■

As with preferences, the value of a tuple may evolve as the user interacts with the system. Three cases can also be considered here.

Static: In this case, the value of a tuple t is pre-calculated (statically) independently of the initial query Q, or subsequent relaxed queries Q'. The relaxation suggestions try to lead the user to a leaf-node that has the highest cost (cost of a non-empty leaf is the maximum value of the tuples that represent that leaf[1])[2]. One can see that this is equivalent to guiding the users to the most-valued tuples. In such cases, the cost of a choice node is computed using Equation 3.3, by setting $C_1 = 0$. Finally, as the optimization objective is to maximize cost, then the cost of a relaxation node is the *maximum* cost of its children.

Semi-Dynamic: In this case, the value of a tuple t is calculated using the initial query Q, the first time it appears in the tuple space of a relaxation. Typical examples of such values are *relevance* score of the tuple to the initial query (here value is same as preference). This computed value of t is reused in all subsequent relaxations. The rest of the process is similar to that of Static.

Dynamic: In this case, the value of a tuple t at a relaxation node is calculated using the latest relaxed query Q' that the user has accepted. This value computation is fully dynamic, and the value of the same tuple t may change as the last accepted relaxed query

[1] Other aggregation functions (such as average) are also possible; the appropriate choice of the aggregation function is orthogonal to our problem.

[2] Cost of an empty-leaf node is 0.

changes. An example of such dynamically changing values are *relevance* of the tuple to the *most recent relaxed query*. Such dynamic value computation approach could be used inside the framework with the optimization objective of *minimizing user effort*, as it minimizes the expected number of interactions. In this case, any leaf node (empty or non-empty) has equal cost of 0. The cost of a choice node is computed using Equation 3.3, by setting $C_1 = 1$ (incurs additional cost of 1 with one more interaction). Finally, if the cost of a relaxation node is computed as the *minimum* cost of its children, then the underlying process will suggest relaxations that terminate this interactive process in the minimum number of steps in an expected sense, thus minimizing the user effort.

More Complex Objective Functions: Interestingly, the proposed framework could even be instantiated with more complex objective functions, such as those that represent a combination of the previous optimization objectives of relevance, price, user effort, etc. (e.g., most relevant results as quickly as possible, or cheapest result as quickly as possible). In such cases, the cost of a leaf node needs to be modeled as a function that combines these underlying optimization factors. After that, the cost computation of the relaxation nodes or the choice nodes in the relaxation tree would mimic either Semi-Dynamic[3] or Dynamic, depending upon the specific combined optimization objective. Further discussion on complex objective functions is out of the scope of this work.

3.4.5 Cardinality constraint

In several applications, the users are interested in non-empty answers that contain a certain minimum number of tuples (specified by some cardinality constraint). Our framework assumes that the user is interested in at least one result. To constrain the number of tuples returned, we simply consider empty any query that returns less than the required cardinality. No other adjustments are needed.

3.4.6 Top-k relaxations

In certain applications, it may be disappointing for the user to get just one relaxation suggestion at a time. Our proposed framework can be readily extended to suggest a ranked list of k relaxations at a given interaction, by suggesting to the user the k best

[3]choice node and relaxation node cost of Static is same as that of Semi-Dynamic.

sibling relaxation nodes based on the cost at a given level in the relaxation tree. However, generating more than one relaxation at the time results in larger computation times, because we need to explore more relaxation alternatives simultaneously. Section 3.5.3 shows how to efficiently modify algorithms that propose only the single best relaxation to obtain the desired results without substantially changing the framework.

3.5 Algorithmic Solution

3.5.1 FullTree

Algorithm 1 FullTree

Input: Query Q
Output: Relaxation Cost of Q

```
 1: T ← ConstructQRTree(Q)
 2: return CompOptCost(T)

 3: procedure ConstructQRTree(Query Q)
 4:     n_relax ← new RelaxationNode(Q)                          ▷ construct the root
 5:     C={c | c∈Constrs(Q) ∧ c is hard}
 6:     ¬Q ← new Query(C)
 7:     if D_¬Q=∅∨Q = ∅ then                                     ▷ non-relaxable query
 8:         return n_relax
 9:     for c ∈ Constrs(Q) do
10:         if c is not hard then
11:             n_resp ← new ChoiceNode()
12:             n_relax.addChild(n_resp)
13:             Q_yes ← new Query(Constrs(Q)\{c})                ▷ remove c
14:             n_resp.yesChild ← (ConstructQRTree(Q_yes))
15:             c_h ← Hard(c)                                    ▷ c_h is the hard version of c
16:             Q_no ← new Query((Constrs(Q)\{c}) ∪ {c_h})
17:             n_resp.noChild ← (ConstructQRTree(Q_no))
18:     return n_relax

19: procedure CompOptCost(Node n)
20:     if n has no children then
21:         return 0
22:     if n is a ChoiceNode then
23:         Cost_yes ← CompOptCost(n.yesChild)
24:         Cost_no ← CompOptCost(n.noChild)
25:         Q_yes ← n.yesChild.Query                             ▷ query in the "yes" child
26:         P_no ← relPref_no(n.Query, Q_yes)                    ▷ Equation (3.1)
27:         P_yes ← 1 − P_no
28:         return P_yes ∗ (C1+Cost_yes)+P_no ∗ (C1+Cost_no)     ▷ Equation (3.3)
29:     else if n is a relaxation node then
30:         return  optimum  CompOptCost(c)
                    c∈n.Children
```

Given Equation 3.4, one can visit the whole query relaxation tree in a depth-first mode and compute the cost of the nodes in a bottom-up fashion. This algorithm is referred to as FullTree. Its steps are described in Algorithm 1. Note that the specific approach has the limitation that the whole tree needs to be constructed first by the procedure ConstructQRTree, and then traversed, a task that is computationally expensive since the size of the tree can be exponential in the number of the constraints in the query. Procedure ConstructQRTree constructs the whole tree starting on the root node representing input query Q, and then it recursively adds child nodes until the query in the node is non-empty or cannot be relaxed further (i.e., no more constraints to relax). Furthermore, for every positive response that the user provides to a relaxation request, the algorithm has to call the database to evaluate the relaxed query. Additionally, based on the specific score computation approach, for every response, it may have to make additional calls to recompute the *prior* and the *pref* value for the tuples in the relaxed query tuple space. This may lead to time complexity prohibitive for many practical scenarios.

3.5.2 FastOpt

To avoid computing the whole query relaxation tree, for each relaxation, we can compute an upper and a lower bound of the cost of its children. From the ranges of the costs that the computation provides, we can identify those branches that *cannot* lead to the branch with the optimal cost. When the specific optimization minimizes the cost (i.e., effort), these are the branches starting with a node that has as a lower bound for its cost a value that is higher than the upper bound of the cost of another sibling node. Similarly, when the objective is to maximize the cost (i.e., lead user to most relevant answers/answers with highest static score), the branches starting with a node that has as a upper bound for its cost a value that is smaller than the lower bound of the cost of another sibling node could be ignored. By ignoring these branches the required computations are significantly reduced. We refer to this algorithm as FastOpt. Algorithm 2 shows the FastOpt pseudocode.

Instead of creating the whole tree, FastOpt starts from the root and proceeds in steps. In each step, it generates the nodes of a specific level. A level is defined here as all the *choice* nodes found at a specific (same) depth, alongside the respective *relaxation*

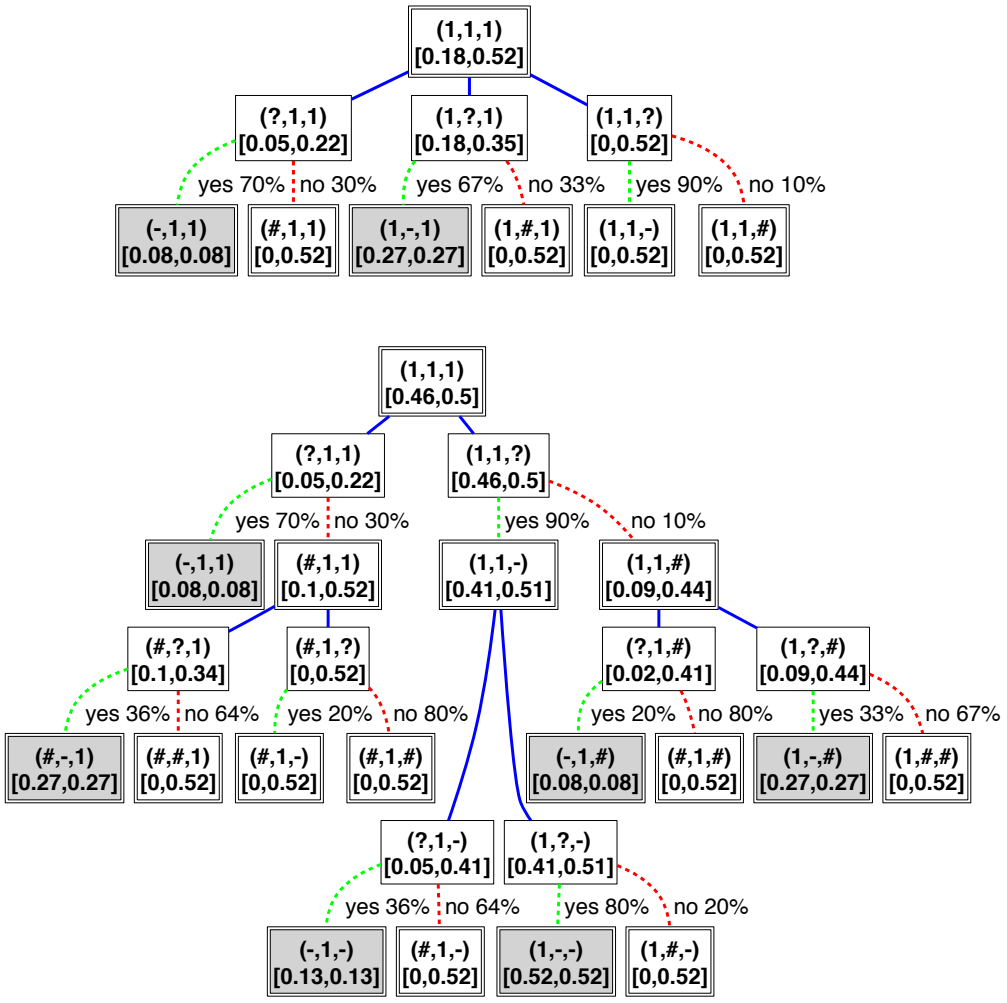


Figure 3.4: Example 3.5 Query Relaxation Tree after 1st and 2nd expansions

nodes they have as children. For the latter it computes a lower and upper bound of their cost and uses them to generate a lower and upper bound of the cost of the choice nodes in that level. When the cost is to be minimized (maximized), those choice nodes with a lower bound higher than the upper bound (or respectively, with an upper bound lower than the lower bound) of a sibling node are eliminated along with all their subtree and not considered further. The computed upper and lower bounds of the choice nodes allow the computation of tighter upper and lower bounds for their parent relaxation nodes (compared to bounds that have already been computed for them in a previous step). The update of these bounds propagates recursively all the way to the root. If a relaxation node models a query that generates a non-empty answer, then it does not expand to its sub-children. Furthermore, after $|\mathfrak{Constrs}(Q)|$ repetitions, the maximum branch length is reached and the relaxation sequence with the optimum cost can be

decided.

The upper and lower bounds of the cost of a node are computed by considering the worst and best case scenario, and depends upon the specific score computation approach. Recall that the cost of a node is computed according to Equations (3.3) and (3.4). When the process seeks to optimize the cost using Semi-Dynamic or Static score computation approach (corresponds to maximum relevance/maximize static score), the lowest cost of a node n at a level L could be as small as 0, because the remaining $|\mathfrak{Constrs}(Q)| - L$ relaxations accepted by the user may have a very small (almost zero) associated probability, resulting in the expected cost to be close to 0. This yields a lower bound $n.LB$=0. Alternately, the highest cost of a node n at a level L is achieved when the user is lead to the highest cost leaf with a "yes" probability of 100% immediately in the very next interaction. This yields an upper bound $n.UB$ = maximum cost of any leaf.

In contrast, when the Dynamic score computation approach is used (corresponds to minimum effort objective), the lowest cost of a node n at a level L of the tree is achieved when the probability for the yes branch of the choice node is 100% and the $Cost(n_{yes})$ in Equation (3.3) is 0. This yields a lower bound $n.LB$=0. Similarly, the highest cost is achieved when all the remaining $|\mathfrak{Constrs}(Q)| - L$ negative responses have a probability of 100%. This yields an upper bound $n.UB = |\mathfrak{Constrs}(Q)| - L$.

At the end, when the computation reaches a level equal to the number of constraints in the query, $|\mathfrak{Constrs}(Q)|$, there is only one choice node to choose. Note that for the leaf nodes of the full tree, the upper and lower bounds coincide.

FastOpt is applicable to *any* cost function for which upper and lower bounds of the cost of a node can be computed even after *only part of the tree* below the node has been expanded. The efficiency of the algorithm relies on whether very tight bounds can be computed even after only a small part of the tree has been expanded. The cost function should also have the following monotonic property: the upper and lower bound calculations should get tighter if more of the tree is expanded. All aforementioned cost functions satisfy this property.

Example 3.5. *Consider the running example in Section 3.3, with the initial query (ABS, DSL, Manual), which aims to guide the user towards cheap cars. The value of a tuple is inversely proportional to its price. Let the normalized values for those tuples be*

Algorithm 2 FastOpt

Input: Query Q
Output: Relaxation Cost of Q

```
1: NextBranch ← new RelaxationNode(Q)
2: NextBranch.L ← 1
3: 𝒯 ← NextBranch
4: repeat
5:     Build tree at level NextBranch.L
6:     N_rlx ← relaxation nodes in level NextBranch.L
7:     for each n∈N_rlx do
8:         if n is a leaf then                          ▷ i.e., no further relaxation is possible
9:             n.UB,n.LB ← 0
10:        else
11:            n.LB ← 0                                 ▷ best case: 100% prob."yes" answer
12:            n.UB ← |Constrs(Q)| − L                  ▷ worst case: 100% prob. "no" answer
13:    UpdateNodes(NextBranch.L)
14:    Prune (𝒯)
15:    𝒯_c ← 𝒯.Children                                  ▷ children of the root node
16:    NextBranch.L ← L + 1
17:    for each n ∈ 𝒯_c do
18:        if n.L = |Constrs(Q)| then
19:            𝒯_c ← 𝒯_c \ {NextBranch}
20:    NextBranch ← arg min_{n∈𝒯_c} {n.UB - n.LB}       ▷ minimum diff ub-lb strategy
21: until NextBranch not NULL
22: return CompOptCost(𝒯)

23: procedure UpdateNodes(Level L)
24:    𝒩_L ← nodes at level L
25:    for each n∈𝒩_L do
26:        if n is a choice node then
27:            Compute probabilities as in Algorithm 1
28:            r_yes, r_no ← "yes" and "no" children of n
29:            n.LB ← P_yes * (C1 + r_yes.LB) + P_no * (C1 + r_no.LB)
30:            n.UB ← P_yes * (C1 + r_yes.UB) + P_no * (C1 + r_no.UB)
31:        else if n is a relaxation node then
32:            n.UB ← optimize_{c∈n.Children} (n.UB)
33:            n.LB ← optimize_{c∈n.Children} (n.LB)

34: procedure Prune(Node r)
35:    if exists n ∈ r.Siblings s.t. r.LB > n.UB then   ▷ r.UB < n.LB if the obj. is to maximize
36:        n.Father.Children ← n.Father.Children \{r}
37:    else
38:        for each n ∈ r.Children do
39:            Prune(n)
```

0.27, 0.08, 0.52, *and* 0.13. *The objective is to select the relaxation node with the highest cost (i.e., expected value).*

Consider Figure 3.4. At the beginning the root node that is created represents the original query with 3 *conditions. Then, in the first iteration (L=*1*), the* 3 *possible choice nodes*

(corresponding to the 3 *attributes of the query) along with their yes and no relaxation child nodes, will be generated (upper half of the figure). Since the relaxation nodes (-,1,1) and (1,-,1) give non-empty answers, they get lower bound (and upper bound) costs of .08 and 0.27 respectively (in the figure, the bounds of every node are denoted in square brackets "*[...]*"). The rest of the relaxation child nodes will be assigned a lower bound of 0 and an upper bound of* 0.52 *(price of the most expensive tuple in the database). Then the bounds of the choice nodes will be updated based on the expected value (considering respective preference probabilities), and the lower bound (resp. upper bound) of the root node will also be updated with the maximum of lower bound (resp. upper bound) cost of its child nodes. In the figure, the values of the quantities* $relPref_{no}(Q, Q')$ *and* $relPref_{yes}(Q, Q')$ *are illustrated next to the label of the no and yes edges, respectively.*

Let us now consider the expansion of the second level. For brevity, we only expand the first and the third child, as shown in the lower half of Figure 3.4. The newly generated relaxation nodes have new upper bounds, apart from those generating empty answers (or cannot be relaxed further) that have a 0 upper bound. This impacts the relaxation nodes of the previous (first) level, whose bounds are updated to [0.08,0.08], [0.1,0.52], [0.41,0.51] and [0.09,0.44]. The updates propagate all the way to the top. Notice that the first child of the root has now an upper bound (0.22) that is smaller than the lower bound of the third child (0.46), thus the first child is pruned and will not be considered further. ■

To further optimize the algorithm, we expand at each step only the node that has the tightest bounds, i.e., the smallest difference between its lower and upper bounds. The intuition is that the difference between these two values will become tighter (or even 0), and the algorithm will very soon decide whether to keep, or prune the node, with no effect on the optimal cost of the tree.

3.5.3 FastOpt for top-k

Unlike the FullTree algorithm that constructs the entire tree and returns all the possible relaxations, the FastOpt does not guarantee to maintain at least k branches for each level. It computes the worst and best case by means of bounds and prunes a branch if it does not participate for sure to the optimal solution. This condition has to be

Algorithm 3 FastOpt for top-k

Input: Query Q, number of relaxations k
Output: Relaxation Cost of Q

```
    ⋮
34: procedure PRUNE(Node r)
35:     P ← {n} ∪ r.Siblings
36:     kUB ← kth highest UB
37:     for i = 1...k, p_i ∈ P do
38:         if p_i.LB > kUB then
39:             P ← P \ {p_i}
40:     for each p ∈ P do
41:         PRUNE(p)
```

adapted to the top-k scenario. In the case of top-k relaxations, a priority queue P of k best children has to be maintained for each subtree. A node can be safely pruned if there exist k elements with a lower (upper) bound greater (lower) than the kth biggest (smallest) upper bound.

Algorithm 3 shows the PRUNE procedure of FastOpt adapted for top-k case with Dynamic objective. All other objectives are easily computed with minimal changes in the code. The algorithm first computes a priority queue P containing sibling nodes in decreasing order of lower bound and the kth biggest lower bound (Lines 34-35). Then it removes from P any node with a lower bound greater than the kth biggest lower bound (Lines 37-39). Finally, the PRUNE procedure is called on the remaining nodes in P. It is easy to see that the algorithm preserves the completeness of the solution, given that the pruned nodes are those that for sure cannot eventually become part of the final solution. The performance of Algorithm 3 is clearly affected by the value k: a large k diminishes the chances to prune branches in advance. We show the relation between k and time in the experimental section.

3.5.4 Approximate Algorithms

3.5.5 CDR

Although the FastOpt algorithm discussed in the previous section generates optimum-cost relaxations and builds the relaxation tree on demand, the effectiveness of this algorithm largely depends on the cost distribution properties between the participating nodes. In the worst case, the FastOpt may still have to construct the entire tree first,

even before suggesting any relaxation to the user. In fact, due to the exponential nature of the relaxation tree, even the FastOpt algorithm may be slow for an interactive procedure for queries with a relatively large number of constraints. Applications that demand fast response time (such as, online air-ticket or rental-car reservation systems) may not be able to tolerate such latency. On the other hand, these applications may be tolerant to slight imprecision. Thus, we propose a *novel approximate solution* that we refer to as the CDR (Cost Distribution Relaxation) algorithm. Like FastOpt, Algorithm CDR also constructs the query relaxation tree on demand, but the constructed part is *significantly smaller*. This is possible because it leverages the distributional properties of the nodes of the tree to probabilistically compute their cost. Of course for applications that are less tolerant to approximate answers, FastOpt may be more desirable, even though the response time may be higher.

Given a query Q, the algorithm CDR computes first the exact structure of the relaxation tree up to a certain level $L < |\mathfrak{Constrs}(Q)|$. Next, it approximates the cost of each L-th level choice nodes by considering the cost distributions of its children and proceeds with the bottom-up computation of the remaining nodes until the root. At the root node, the best relaxation child node is selected, and the remaining ones are pruned. Upon suggesting this new relaxation, the algorithm continues further based on the user's response. There are three main challenges in the above procedure: (i) in the absence of the part of the tree below level L, how will the cost of level L nodes be approximated? (ii) How is the cost of the intermediate nodes approximated in the bottom-up propagation? and (iii) how is the best relaxation at the root selected? To address these challenges, we propose the use of the distributional properties of the cost of the nodes and the employment of probabilistic computations, as described next.

3.5.5.1 Cost Probability Distributions Computation

The algorithm computes the distribution of the cost of the nodes at level L (first the relaxation nodes, then the choice nodes), and higher by assuming that the underlying distributions are independent and by computing the *convolutions* [ADGK09] of the *probability density functions* (pdf for short).[4] We adopt convolution of distributions definitions from previous work [ADGK09] to compute the probability distribution of the

[4]The independence assumption is heavily used in database literature, and as the experimental evaluation shows, it does not obstruct the effectiveness of our approach.

cost of the nodes in the partially built relaxation tree, as defined below. Then, in Section 5.2, we discuss how such convolution functions could be efficiently approximated using histograms.

Definition 3.4 (Sum-Convolution of Distributions)**.** Assume that $f(x)$, $g(x)$ are the pdfs of two independent random variables X and Y respectively. The pdf of the random variable $X+Y$ (the sum of the two random variables) is the convolution of the two pdfs: $*(\{f,g\})(x) = \int_0^x f(z)g(x-z)\,dz$.

Definition 3.5 (Max-Convolution of Distributions)**.** Assume that $f(x)$, $g(x)$ are the pdfs of the two independent random variables X, Y respectively. The pdf of the random variable $Max(X,Y)$ (the maximum of the two random variables) is the max convolution of the two pdfs:$_{max} * (\{f,g\})(x) = f(x)\int_0^x g(z)\,dz + g(x)\int_0^x f(z)\,dz$.

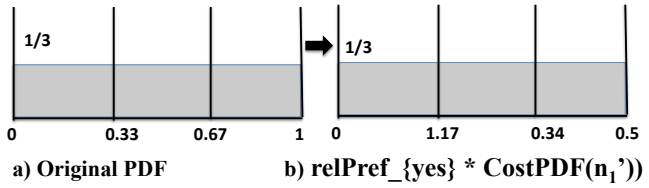
The Min-Convolution can be analogously defined, and moreover these definition can be easily extended to include more than two random variables.

We now describe how to estimate the cost distribution of each node using Sum convolution and Max(similarly Min) convolution. We denote as $CostPDF(n)$ the probability density function of the cost of a node n.

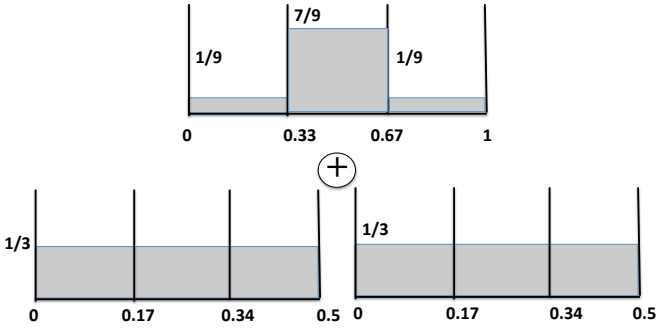
Cost distribution of a Relaxation Node: We first need to compute the cost distribution of nodes at level L and then propagate the computation to the parent nodes. We consider the pdf of each node at level L to be *uniformly distributed* between its upper and lower bounds of costs as described in FastOpt.

For relaxation nodes at higher levels, we need to compute the optimum cost over all the children nodes. Note that, optimization objectives associated with Semi-Dynamic and Static require Max-convolution as the score of the relaxation nodes are maximized in those cases. In contrast, Dynamic could be used to minimize effort - requiring Min-convolution to be applied to compute the minimum cost of the relaxation nodes.

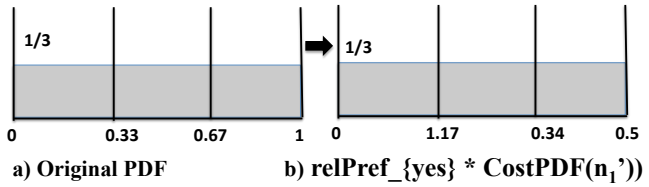
Cost distribution of a Choice Node: The computation of the cost distribution involves the summation operation between two pdfs ($CostPDF(n_{yes})$ and $CostPDF(n_{no})$), and between a constant and a pdf (e.g., $C1 + CostPDF(n_{yes})$) (ref. equation (3.3)). Assuming independence, the former operation involves the sum convolution of two pdfs, whereas the latter requires the sum convolution between a pdf and a constant. In addition, $C1 + CostPDF(n_{yes})$ (similarly $C1 + CostPDF(n_{no})$) needs to be multiplied



(a) "yes" branch



(b) choice node



(c) non-leaf relaxation nodes

Figure 3.5: $CostPDF(n)$ for (a) the "yes" branch of a choice node, (b) choice node, and (c) non-leaf relaxation nodes.

with a constant $relPref_{yes}$ (similarly $relPref_{no}$). We note this multiplication operation between a constant and a pdf can be handled using convolution as well.

Selecting Relaxation at the Root: Given that the root node in the relaxation tree contains k children, the task is to select the best relaxation probabilistically. For each child node n_i of root with pdf $CostPDF(n_i)$, we are interested in computing the probability that the cost of n_i is the largest (resp. smallest) among all its k children when we want to maximize (resp. minimizes) the cost of the root. Formally, the suggested relaxation at the root (N_{rlx}) equals $N_{rlx} = \arg\max n_i(\Pr(Cost(n_i) \geq \prod_{j=1,j\neq i}^{k} Cost(n_j)))$ (respectively $N_{rlx} = \arg\min n_i \Pr(Cost(n_i) \leq \prod_{j=1,j\neq i}^{k} Cost(n_j)))$

Given the user response, the above process is repeated for the subsequent nodes until the solution is found.

Algorithm 4 CDR

Input: Query Q, level L
Output: Relaxation node to be proposed at the root

for $l = 1 \dots L$ **do**
  $N_{rlx} \leftarrow$ relaxation nodes in level l
  for all $n \in N_{rlx}$ **do**
    if n is a leaf **then** ▷ n is non-empty or not relaxable
      $CostPDF(n) \leftarrow$ uniform in $[1, |\mathfrak{Constrs}(Q)| - l]$
for $l = L \dots 1$ **do**
  UpdateRelaxationNodes(l)
  UpdateChoiceNodes(l)
$N_u \leftarrow$ child nodes of root
return $\arg\max_{n \in N_u}(\Pr(Cost(n) \leq \prod_{j=1, j \neq n}^{|N_u|} Cost(n_j)))$

procedure UpdateChoiceNodes(Level l)
  $N_{rsp} \leftarrow$ choice nodes at level l
  for all $n \in N_{rsp}$ **do**
    Compute probabilities as in Algorithm 1
    $r_{yes}, r_{no} \leftarrow$ "yes" and "no" child of n
    $CostPDF(n) \leftarrow P_{yes} * (C1 + CostPDF(r_{yes})) + P_{no} * (C1 + CostPDF(r_{no}))$

procedure UpdateRelaxationNodes(Level l)
  $N_{rlx} \leftarrow$ relaxation nodes at level l
  for all $n \in N_{rlx}$ **do**
    $N_u \leftarrow$ choice child nodes of n
    $CostPDF(n) = \min(CostPDF(n_1), \dots, CostPDF(n_{|N_u|}))$

3.5.5.2 Efficient Computation of Convolutions

The practical realization of our methodologies is based on a widely adopted model for approximating arbitrary pdfs, namely *histograms* (we adopt equi-width histograms, however any other histogram technique is also applicable). In [ADGK09] it has been shown that we can efficiently compute the Sum, Max, and Min-convolutions using histograms to represent the relevant pdfs. In the following example, we illustrate how histograms may be used for representing cost pdfs at nodes of the relaxation tree.

Example 3.6. *Consider a query Q with $|\mathfrak{Constrs}(Q)|$=5 and empty answers, and assume that the approximation algorithm sets $L = 2$. Let us assume that cost is required to be maximized. Consider a choice node n at level 2, which has child relaxation nodes n'_1 (for a positive response to the relaxation proposal) and n'_2 (for a negative response); Wlog, let 1 be the upper bound of cost of n'_1[5]. Thus, the cost of each child is a pdf with uniform distribution between 0 and 1. $CostPDF(n'_1)$ is approximated using a 3-bucket equi-width histogram, and if we assume that $relPref_{yes}$ and $relPref_{no}$ of n are 0.5, the*

[5]Recall that the lower bound is always 0.

$CostPDF(n)$ can be computed using Equation 3.3 by approximating the pdf of the cost of each child with a 3*-bucket histogram. Figures 3.5 (a) - (b) illustrate these steps.*

The algorithm continues its bottom-up computations, and considers relaxation nodes in the next higher level: at level 1*, given a relaxation node (n') that has k children (each corresponds to a choice node in level* 2*), $CostPDF(n')$ is computed by using Equation 3.4 and applying max-convolution on its children (see Figure 3.5 (c)). Once the pdf of the cost of every relaxation node at level* 1 *has been determined, the algorithm next computes the pdf of cost of each level* 1 *choice nodes using sum-convolution, and so on.*

3.5.6 FastCDR

Completely constructing L levels as required by the CDR when it needs to return top-k relaxations becomes computationally expensive. Since the FastOpt behaves like the FullTree when k is close to $|\mathfrak{Constrs}(Q)|$ we need a different algorithm to efficiently solve the problem while guaranteeing quality close to optimal.

CDR can be further optimized by removing from the search space non promising branches in the first L levels, and continuing the exploration over the remaining nodes. This can be naturally achieved using FastOpt in the first L levels and then expandingthe remaining tree using the CDR. Clearly, the solutions produced applying this strategy are as good as those of CDR, and hopefully better if the removed branch is the one that CDR would select for expansion. Furthermore, if the optimal solution is found within the first L levels, the time performance will be the same as FastOpt, eliminating the requirement of CDR that further expands one of the branches if the optimal is not detected in the first iteration. We refer to this new hybrid algorithm as Fast Cost Distribution Relaxation (FastCDR, in short). Algorithm 5 describes its steps in pseudocode. It works exactly as the CDR except that it first generates all the candidate nodes using FastOpt for top-k (see Line 12-26).

3.6 Extensions

In this section we present extensions to the framework that allow the use of our methods in non-boolean attributes. Moreover, we describe some strategy to explore other

Algorithm 5 FastCDR

Input: Query Q, level L, number of relaxations k
Output: top-k relaxations at root

GenerateNodes(L)
UpdateRelaxationNodesLowest(L)
for $l = L..1$ **do**
    UpdateRelaxationNodes(l) ▷ see Algorithm 4
    UpdateChoiceNodes(l) ▷ see Algorithm 4
$N_u \leftarrow$ child nodes of root
P $\leftarrow N_u$ ordered by $\Pr(Cost(u) \leq \prod_{\forall j=1, j\neq u}^{|N_u|} Cost(j))$
return $p_1, ..., p_k \in$ P

procedure UpdateRelaxationNodesLowest(Level L)
    $\mathcal{N}_{rlx} \leftarrow$ relaxation nodes at level L
    for all relaxation node $n \in \mathcal{N}_{rlx}$ **do**
        $CostPDF(n) \leftarrow$ uniform in $[1, |\mathfrak{Constrs}(Q)| - L]$

procedure GenerateNodes(Level L)
    for l $= 1 \ldots |L|$ **do**
        $N_{rlx} \leftarrow$ relaxation nodes in level l
        for relaxation node $u \in N_{rlx}$ **do**
            if u represents an empty answer query **then**
                UpdateRelaxationNodesLowest(u)
                $u.UB = u.UB = 0$
            else
                $u.LB = 0$
                $u.UB = |\mathfrak{Constrs}(Q)| - L$
        UpdateChoiceNodes(L)
        for $TLev = (L-1)..0$ **do**
            UpdateRelaxationNodes(TLev)
            UpdateChoiceNodes(TLev)
        Prune(L) ▷ see Algorithm 2

alternatives if the process ends in an empty-answer.

3.6.1 Databases with categorical and numerical attributes with hierarchies

Our framework can ingest categorical data, provided that the data is organized and stored in a specific format. If not, a preprocessing step needs to be executed to bring the data in that format, an then the algorithm can run as in the boolean database case.

Categorical attributes. The categorical data reorganization is performed attribute by attribute. Each value of a categorical attribute is an attribute itself in a boolean database. For instance, attribute *price*={*low, average*} is represented as two attributes *price-low* and *price-high* in the corresponding boolean database and a tuple has value

1 on *price-low* if the value of the attribute *price* is *low*. Given a query on categorical attributes, we translate it into boolean using the converted attributes. The rest of the computations in the framework remain unchanged.

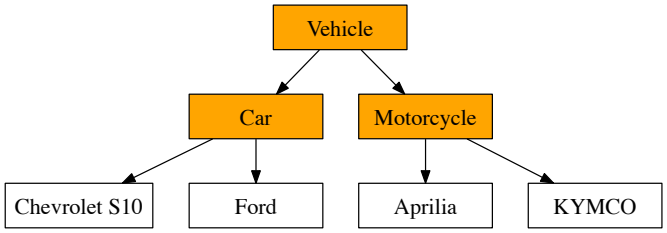
The framework can also accommodate the case in which an order or a preference is induced on the categorical attribute values, by changing the $pref$ function to steer the user towards a specific attribute value.

Hierarchical attributes. The same idea applies to conceptual hierarchies (i.e., hierarchies that have a partial or total order of the attributes). In other words, we expand the query lattice (refer to Figure 3.2) using the hierarchies of each attribute, construct the query relaxation tree, and finally compute the cost. Additional priorities on the hierarchy attributes may be embedded in the *pref* computation. Figure 3.6 contains an illustration of the categorical data in a boolean database. *Vehicle* contains two categories {*Car, Motorcycle*}. Each of these categories contains the following subcategories {*Chevrolet S10, Ford*} and {*Aprila, KYMCO*}. An attribute is created for each category. Each tuple has value 1 or 0 for an attribute depending on whether the tuple represents an object belonging to that category or not. For instance, the attribute corresponding to category named "car" receives the value 1 for a tuple modeling a "Chevrolet S-10" car (because it belongs to the category "car") and 0 for "motorcycle" (because it does not belong to the category "motorcycle"). The table at the bottom of Figure 3.6 illustrates how a *Ford* car and a *KYMCO* motorcycle are stored in the boolean database.

From an implementation point of view, to navigate through the hierarchies levels during the execution, the algorithm keeps an additional structure that stores information about the categorical attributes. During the interactive process with the user, and when the next best relaxation is computed on the fly, the algorithm will also check if any relaxation can be applied in the hierarchy levels (i.e., if the current attribute is a "car", then it is evaluated whether relaxing one level up to "vehicle" makes sense).

Numerical attributes. The *numerical attributes* are more elastic from a relaxation point of view as the numerical value of an attribute can be extended over large ranges of values. If we consider buckets over data ranges and hierarchies on top of them, this case is reduced to the hierarchical case and the above methods apply.

Our algorithms can equally ingest databases containing all these kinds of attributes



Vehicle	Car	Motorcycle	Chevrolet S10	Ford	Aprilia	KYMCO
1	1	0	0	1	0	0
1	0	1	0	0	0	1

Figure 3.6: Categorical data representation in a Boolean database.

simultaneously. However, we recall that the focus of the current work is not on optimized ways of operating with different types of data, but on the optimization of the interactive process with the user.

3.6.2 Drill down / Roll back

Proposing to the user the relaxations that at any given moment seem to be the most promising (according to the criteria we have already discussed) and the responses that the user has until that moment provided, may end up into a query that no more relaxations are possible or even if there are relaxations, they will not lead into a query with a non-empty answer. This type of relaxations are represented as a leaf in the query relaxation tree. If this happens, then instead of simply terminating the process, which may not be the best option for the user, we can retract one or more relaxations and follow an alternative path, allowing the process to continue.

There are many options that can be used to decide which alternative path to follow. For instance, one could: (i) go 1-relaxation back, and continue with the next best sibling; (ii) go t-relaxations back, and continue with the next best sibling, with t having some predefined value; (iii) go t-relaxations back, and continue with the next best sibling, with t being the minimum number of levels back s.t. the corresponding subtree has at least $m > 0$ tuples, and m being some parameter; (vi) ask the user to which of the attributes she selected so far she is willing to give up; ban that attribute, go back to the level where that attribute was relaxed and continue with the next best sibling; (v) same, but allow the user to select $t > 1$ attributes that she is willing to give up, and

discard all of them at once, then go to the attribute at the highest level in the tree, and recompute from there and by not considering the other $t-1$ attributes relaxations; (vi) same as before, but prioritize all other attributes that are not in the set of attributes under that attribute, and consider the rest only after this set of attributes is exhausted; and as a final alternative, (vii) once we go back to another attribute (selected by one of the previous ways), make zero all probabilities of all attributes the user selected after that attribute, and continue with the next best sibling.

3.7 Experimental Evaluation

We present our experimental evaluation in this section, investigate the effectiveness and scalability of our proposed solutions, and compare our proposed framework with a number of related works and baseline methods. Our prototype is implemented in Java v1.6, on an i686 Intel Xeon X3220 2.40GH machine, running Linux v2.6.30. We report the mean values, as well as the 95% confidence intervals, wherever appropriate.

Datasets. We use two real datasets from diverse domains, namely, used cars and real estate. The Cars dataset is an online used-car automotive dealer database from US, containing 100,000 tuples and 31 attributes. The Homes dataset is a US home properties dataset extracted from a nationwide realtor website, comprising of 38,095 tuples with 18 boolean attributes. Based on the Cars dataset, we also generated datasets ranging between 20,000-500,000 rows (Cars-X), where we maintained the original (multidimensional) distribution of attribute values. This offers a realistic setting for testing the scalability of our algorithms.

Queries. We consider a workload of 20 random empty-answer queries, initially containing only soft constraints. User preferences are simulated using our *relPref* value associated with each choice node.

3.7.1 Implemented Algorithms

Interactive. This algorithm is from our interactive query relaxation framework, and we implement three different instances of the preference computation: (i) Dynamic: a

minimization of the user effort, with parameter $C_1 = 1$ and leaf cost 0; (ii) Semi-Dynamic: a maximization of the answers quality with parameter $C_1 = 0$ and leaf cost equal to the maximum value of the preference function in the result-set; and (iii) Static: a maximization of a randomly chosen static value for each tuple, with parameter $C_1 = 0$ and leaf cost the maximum profit of the result-set. Additionally, we implement FullTree, FastOpt, and our two parameterized algorithms CDR and FastCDR. For the experimental evaluation, we use (Fast)CDR with $L = 3$, and 20 buckets, exibiting the best trade-off in terms of time and quality, as shown in Section 3.7.8. Finally, we implement top-k variants of FastOpt and FastCDR that interactively propose k relaxations at each step.

Baselines. We implement two simple baseline algorithms: Random always chooses the next relaxation to propose to the user at random, and Greedy greedily selects the first encountered non-empty relaxation, considering only the next level.

Related Works. In this group of algorithms we have considered a number of approaches from the related literature.

• top-k: This algorithm takes user-specific ranking functions as inputs (user provides weights for each attribute of the database), and we show the top-k tuples, ranked by the linear aggregation of the weighted attributes.

• Why-Not: This algorithm is from [TC10], non-trivially adapted for the empty-answer problem. We note that method [TC10] is primarily designed for numerical data, and inappropriate for empty-answer problem, since it assumes that the user knows her desirable answer (unlike empty-answer problems). We make the following adaptations: given a query with empty-answers, we apply our relevance-based pref ranking function (Section 3.4) to determine the most relevant tuple in the database that matches the query (non-exact match). We use that tuple as user's desirable answer, then convert the categorical database to a numeric one (in scale $0 - 1$), and apply [TC10] to answer the corresponding Why-Not query. The algorithm generates a set of relaxations that we present to the user.

• Multi-Relaxations: This algorithm is from [Jan07], suggesting all minimal relaxations to the user.

• Mishra and Koudas: This algorithm is from [MK09]. Given a query, the method suggests a set of relaxations, such that the number of tuples in the answer set is bounded

by a user specified input. Our empirical study on the queries presented above exhibits that 81% of the queries with 8 constraints do not lead to a non-empty answer (i.e., failing queries), if the relaxations take place along each attribute independently (and ignore multiple attribute relaxations in conjunction). Our empirical study, depicted in Figure 3.7, shows that 81% of the queries with 8 constraints do not lead to a non-empty answer (i.e., failing queries), if the relaxations take place along each attribute independently (and ignore multiple attribute relaxations in conjunction). Therefore, [MK09] largely fails to successfully address the empty-answer problem at hand.

Summary of Experiments.

We implement the related works discussed above, and set up a user study comparing the related works with our proposed framework in Section 3.7.2. Additionally, we also present other two user studies: a separate study that validates the effectiveness of different cost-functions supported by our framework, and a comparison between single and top-k relaxations produced by our methods. An empirical comparison among the different objectives is presented in Section 3.7.3. Section 3.7.4 presents quality experiments to experimentally demonstrate the effectiveness of our proposed framework in optimizing the preferred cost function (by the cost of the root node of the relaxation tree). Section 3.7.5 presents the scalability studies. Section 3.7.6 and Section 3.7.7 present the results at increasing k and cardinality, respectively. Section 3.7.8 reports the effectiveness of the approximate algorithms CDR and FastCDR varying the parameter (L).

Summary of Results. Our study concludes the following major points - (1) Existing methods are unable to address the same broad range of objectives (e.g., the case when the overall goal conflicts with user preference) as we do. (2) More than 60% of the users prefer "step-by-step" interactive relaxation suggestion to non-interactive top-k results based on user defined ranking functions (11%), or returning all relaxations suggestions in one step [TC10, GS91] (20%). (3) User satisfaction is maximum (i.e., over 90%) with the returned results by our framework even for seller-centric optimization objectives. (4) Our proposed algorithms scale well with increasing dataset or query size (experiments up to 500k tuples). (5) Algorithm CDR can effectively balance between efficiency and the quality of the returned results (within a factor of 1.08 from the optimal). (6) FastCDR preserves quality close to optimal and real-time performance at increasing k.

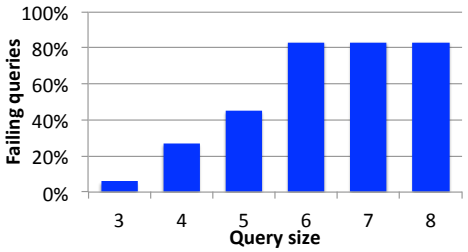


Figure 3.7: Failing queries in Mishra and Koudas vs query size.

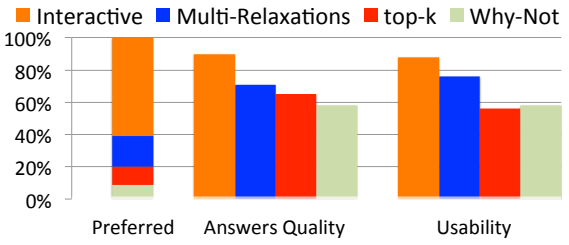


Figure 3.8: Comparison of user satisfaction with different related work.

Q1	Rate the suggested refinements in this interactive process.
Q2	Did you like the system guiding you in the relaxation process?
Q3	Did the system help you arrive to the results fast?
Q4	Did you prefer using the help of this system to relaxing the query by yourself?

Table 3.1: Questions asked in the user-study.

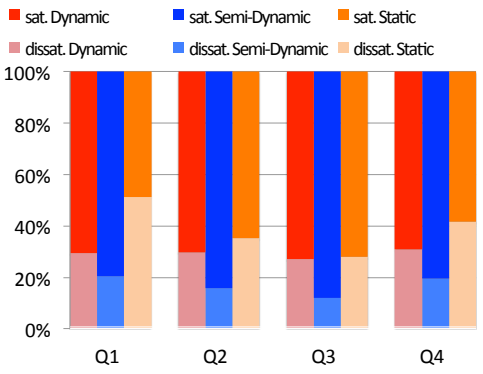


Figure 3.9: Percentage of satisfied and dissatisfied users.

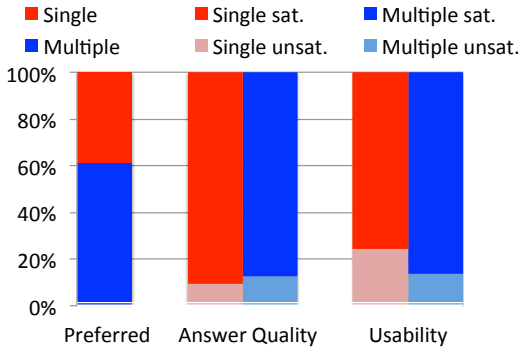


Figure 3.10: Comparisons of single ($k = 1$) and multiple ($k > 1$) relaxations in terms of user satisfaction.

3.7.2 User Study

We build a prototype of our system and use the Homes database to conduct two user studies with Amazon's Mechanical Turk (AMT).

Comparison to previous work. In this user study, we compare our proposed method Interactive (for seller-centric optimization) with top-k, Why-Not and Multi-Relaxations. We hire 100 qualified AMT workers to evaluate 5 different queries, and measure user satisfaction in a scale of 1 to 4 [6] independently and in comparison with other methods. We ask each worker, which method is most preferable (Favored), rate her satisfaction

[6] 1- very dissatisfied, 2 - dissatisfied, 3- satisfied, 4-very satisfied

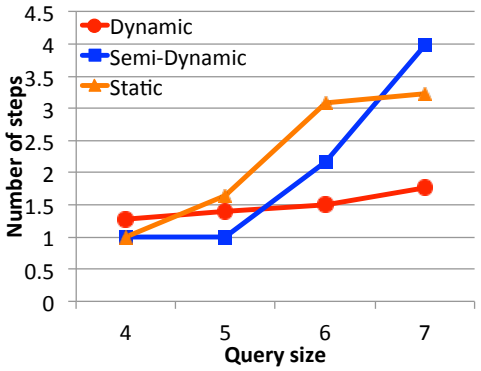


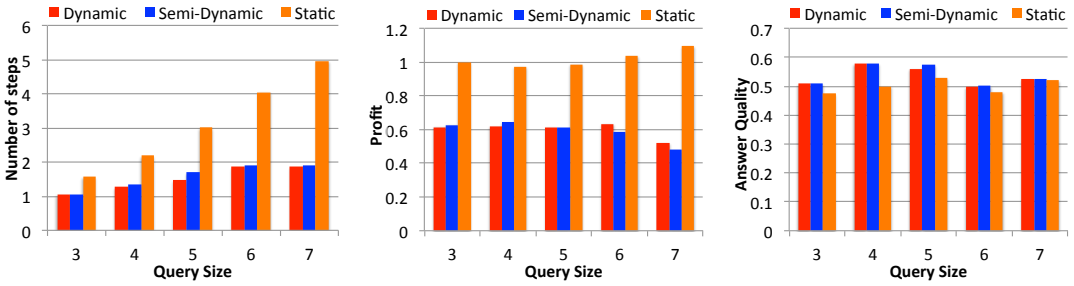
Figure 3.11: Number of steps vs query size in the user study.

with the quality of the returned results (Answers Quality) for each method, and rate her satisfaction with the effectiveness of each of methods (Usability). In addition we ask them the age range and the level of expertise with the use of computers and Internet (naïve to IT professional user in the range 1-4). As depicted in Figure 3.8, more than 60% of the users prefer Interactive compared to other methods, and only 11% of users like to design ranking functions. With regard to result quality, more than 80% think that Interactive is appropriate for obtaining good quality results. At the other extreme, the adaptation of Why-Not algorithms produce good quality results only for 58% of the workers. With regard to method usefulness, the users are asked to independently evaluate the usefulness of each of the four methods in obtaining fast answers. 88% of the users prefer Interactive (i.e., give scores of 3 or 4), whereas 76% prefer Multi-Relaxations, 65% top-k, and 58% Why-Not. Finally, the users are also asked to score Interactive in terms of overall satisfaction: 91% workers are very satisfied with Interactive, out of which 49% are naïve users (data is not shown in Figure 3.8).

Optimization goals comparison. We set up three different tasks, hire a different set of 100 workers to test different optimization functions (without actually knowing them) in our framework. We propose five empty-answer queries per HIT, with $4-7$ attributes. The study uses the FastOpt algorithm, and the workers are asked to evaluate the suggested refinements (Q1), the system guidance (Q2), the time to arrive to the final result (Q3), and the system overall (Q4), in a scale of 1 (very dissatisfied) to 4 (very satisfied). We compare different optimization functions in terms of number of steps, profit, and answer quality (we only show results for the number of steps; the others exhibit a behavior similar to the ones described in the previous section, and are omitted

for brevity). The analysis shown in Figure 3.11 shows that Dynamic guides users to non-empty results 2 times faster than the other approaches when the query size increases. The results (Figure 3.9) also show that the users express a favorable opinion towards our system. As expected, the Static method, being seller-centric, is the least preferred, yet satisfies 60% of the users on an average. The Semi-Dynamic approach is the most preferable overall, producing higher quality results faster, and highest user satisfaction (ranging between 72-89%) .

Top-k quality. In this user study we compare the quality of the FastOpt when the number of relaxations proposed changes. The user does not provide any yes/no answer if multiple relaxations are shown, assuming only yes answers in that case. We ask the users to evaluate 5 different queries with single or multiple (2,3) relaxations proposed. At the end we propose to: choose among single and multiple relaxations (Preferred), evaluate the overall quality of the answers for single and multiple (Answer quality) and, evaluate the easiness of use (Usability) in a scale of 1 to 4. Users found both techniques useful and appreciated their features, with a slight preference for Multiple, mainly because of its small advantage in terms of usability.



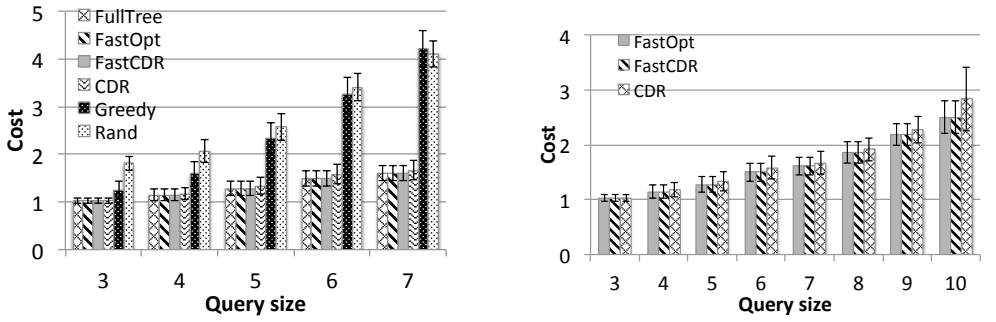
(a) Effort **(b)** Profit **(c)** Quality

Figure 3.12: Comparison of different goals in terms of objective function value (effort, profit, quality) vs query size (Homes dataset).

3.7.3 Preference Computation Comparison

Figure 3.12a shows how different cost functions behave with respect to the number of expected steps before we find a non-empty answer. We notice that the Static approach performs significantly worse than the other two. This is due to the fact that, in order to find more profitable tuples, the best option is to relax several constraints, which leads to producing long optimal paths. On the other hand, Figure 3.12b shows that Static achieves considerably better profit results, which means that the extra cost incurred pays

off. Figure 3.12c measures the quality of the results, and indicates that the inclusion of the preference function in the probability computation tends to favor good quality answers. We also observe that the behavior of Static is very different from that of Dynamic and Semi-Dynamic, since it does not depend on the user preference, while the other two are highly user-centric, thus leading to (slightly) better answer quality.



(a) Proposed algorithms and baselines comparison

(b) FastOpt, (Fast)CDR-3 comparison for big queries

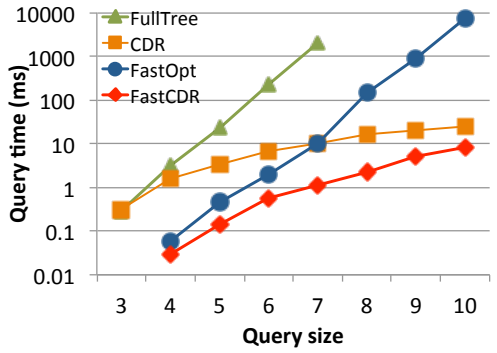
Figure 3.13: Relaxation cost vs query size (Homes dataset).

3.7.4 Effectiveness

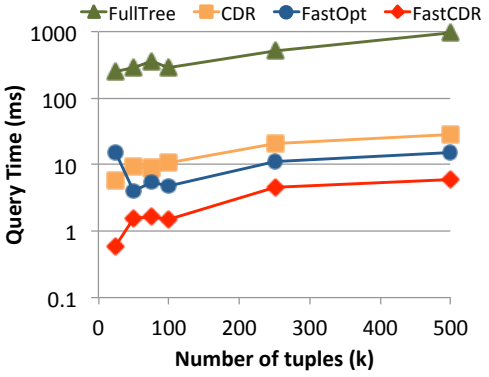
In the next set of experiments, we evaluate the effectiveness of the algorithms by measuring the cost of the relaxation for different query sizes. For brevity, we present only the results for Dynamic, since there are no significant differences among those objectives and we compare algorithm based on the cost function value at the root (see Equation 3.3). In these experiments, we use queries of size up to 7, because this is maximum possible size for running FullTree in our experimental setup. Figure 3.13a depicts the results for the Homes dataset (normalized by the cost of FullTree for query size 3). The Cars dataset results are similar, and we omit them for brevity.

The graph confirms the intuition that the Random and Greedy algorithms are not able to find the optimal solution (i.e., the solution with the minimum expected number of relaxations). In addition, their relative performance gets worse as the size of the query increases, since the likelihood of making non-optimal choices increases as well. For query size 7, Random produces a solution that needs 2.5 times more relaxations than the optimal one and 2.6 for Greedy. As expected, they also exhibit the largest variance in performance.

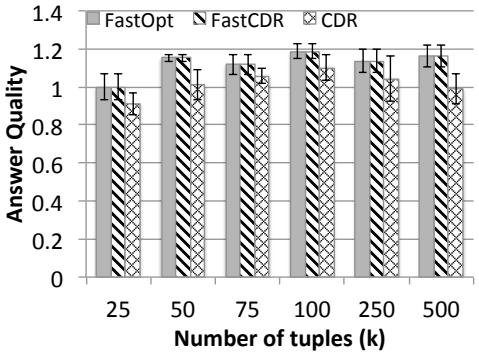
On the other hand, CDR and FastCDR perform very close to FullTree, choosing the best path in most of the cases. The same observation holds for larger queries (upto 10 attributes, refer to Figure 3.13b), where all values are normalized by the cost of FastOpt for query size 3). Our results also shows that CDR remains within a factor of 1.08 off the optimal solution (expressed by FastOpt in the graph) corroborating its effectiveness to the empty-answer problem. Interestingly, FastCDR, due to the pruning strategy of FastOpt, guarantees nearly optimal answers even for large queries. The results demonstrate that CDR and FastCDR are effective solutions to the empty-answer problem, even when the query size grows much larger than L (set to 3 for all the experiments).



(a) Query time (log scale) vs query size (Homes dataset).



(b) Query time vs dataset size, for query size 6 (Cars-X datasets).



(c) Relaxation cost vs query size, for FastOpt, (Fast)CDR-3 (Homes dataset).

Figure 3.14: Results with increasing query size.

3.7.5 Scalability

Figure 3.14 illustrates experiments on the scalability properties of the top performers, FullTree, FastOpt, CDR and FastCDR, when both the size of the query and the size of

the database increase.

Figure 3.14a shows the time to propose the next relaxation, as a function of the query size. We observe that the FastOpt algorithm performs better than CDR when the query size is small (i.e., less than 8), but worse than FastCDR that combines the characteristics of both. This behavior is explained by the fact that CDR is always computing all the nodes of the relaxation tree up to level L, while FastCDR applies pruning rules to speed up the computation. In contrast, FastOpt is able to prune several of these nodes, leading to a significantly smaller tree. For query sizes larger than 8, CDR and FastCDR compute more than two order of magnitudes less relaxation nodes than FastOpt. Moreover, as the query size increases CDR performs close to FastCDR since the time to compute the levels below L dominates the computation. Finally, FullTree has an acceptable performance only for small query sizes (in our experimental setting we could only execute FullTree for query sizes up to 7).

The FastOpt algorithm remains competitive to CDR for small sized queries, but becomes extremely slow for large query sizes, requiring almost 10 sec for queries of size 10. For the same queries, FastCDR executes three orders of magnitude faster, requiring 8.3 ms to produce the next relaxation, and significantly less than 1 ms for smaller queries.

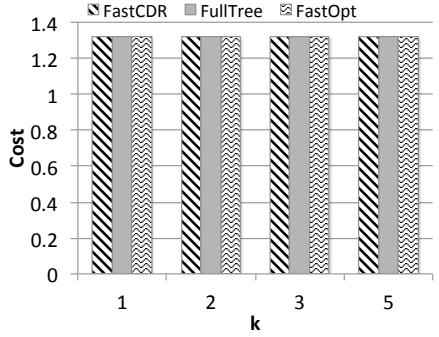
We also experiment with varying dataset sizes between 25K-500K tuples, using the Cars-X datasets, having query size set to 6. Figure 3.14b indicates that query time is moderately affected by size of the database, since relaxation tree becomes smaller with increasing dataset size even though the database access time increase. This happens because more tuples in the dataset translate to an increased chance of a specific relaxation (i.e., node in the query relaxation tree) being non-empty. Note that, even though CDR involves more nodes than FastOpt, it performs similarly, since FastOpt has to build the entire tree before producing the first relaxation, whereas, CDR chooses the best candidate relaxation after computing the first L levels of the tree, which translates to reduced time requirements per iteration.

We also show in Figure 3.14c the impact to the answer quality (Semi-Dynamic preference computation) as a function of database size. As expected, we obtain more qualitative answers with bigger databases, since the likelihood of having non-empty queries with good results also increases. As per quality comparison, FastCDR performs close to FastOpt with different optimization criteria.

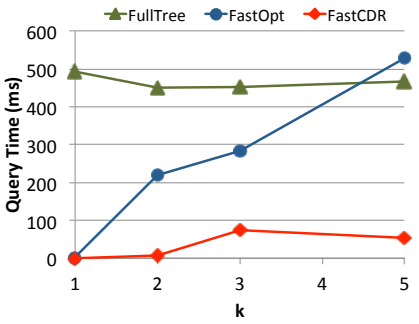
Overall, we observe that FullTree quickly becomes inapplicable in practice, while FastOpt is useful only for small query sizes. In contrast, CDR and FastCDR are able to propose relaxations in less than 10 ms, even for queries with 10 attributes, and always produces a solution that is very close to optimal.

3.7.6 Impact of k in the top-k relaxations

In the next set of experiments, we show the impact of the k in the top-k relaxations. The FastOpt algorithm is compared with the FastCDR in terms of time and quality. Figure 3.15b shows the query time of each method at increasing k varying the number of reformulations and averaging the results obtained with query size 3-7. FastCDR takes nearly constant time in k and returns results one order of magnitude faster than the optimal. On the other hand, FastOpt query time increases linearly with k performing worse than FullTree with $k = 5$. The reason is that the pruning power of FastOpt is affected by k, since the number of branches in the relaxation to be constructed is bigger. Moreover, the overhead induced by the pruning procedure negatively affects the query time. In terms of quality, Figure 3.15a shows that FastCDR is approximately as good as optimal at increasing k.



(a) Cost vs number of relaxations

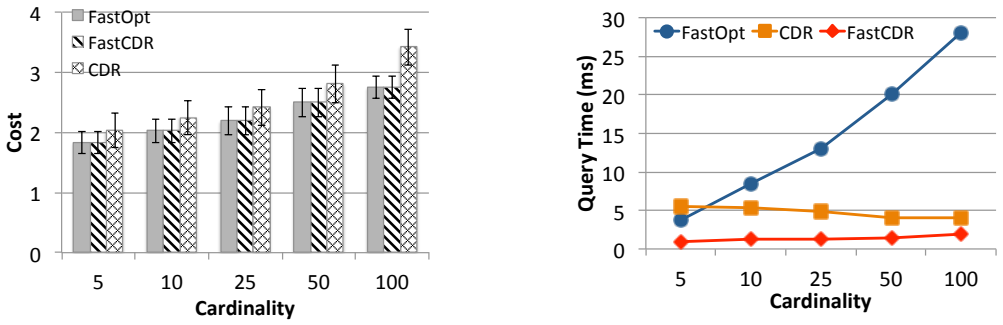


(b) Time vs number of relaxations

Figure 3.15: CDR and FastCDR behavior with increasing number of relaxations proposed at each step.

3.7.7 Cardinality Impact

We analyze the impact of cardinality on our methods. A high cardinality value tends to produce deeper trees, affecting time and quality. Figure 3.16b shows that FastOpt takes 6 times more when the cardinality moves from 5 to 100. The approximate algorithms CDR



(a) Cost vs cardinality

(b) Time vs cardinality

Figure 3.16: CDR behavior with increasing cardinality in Homes dataset.

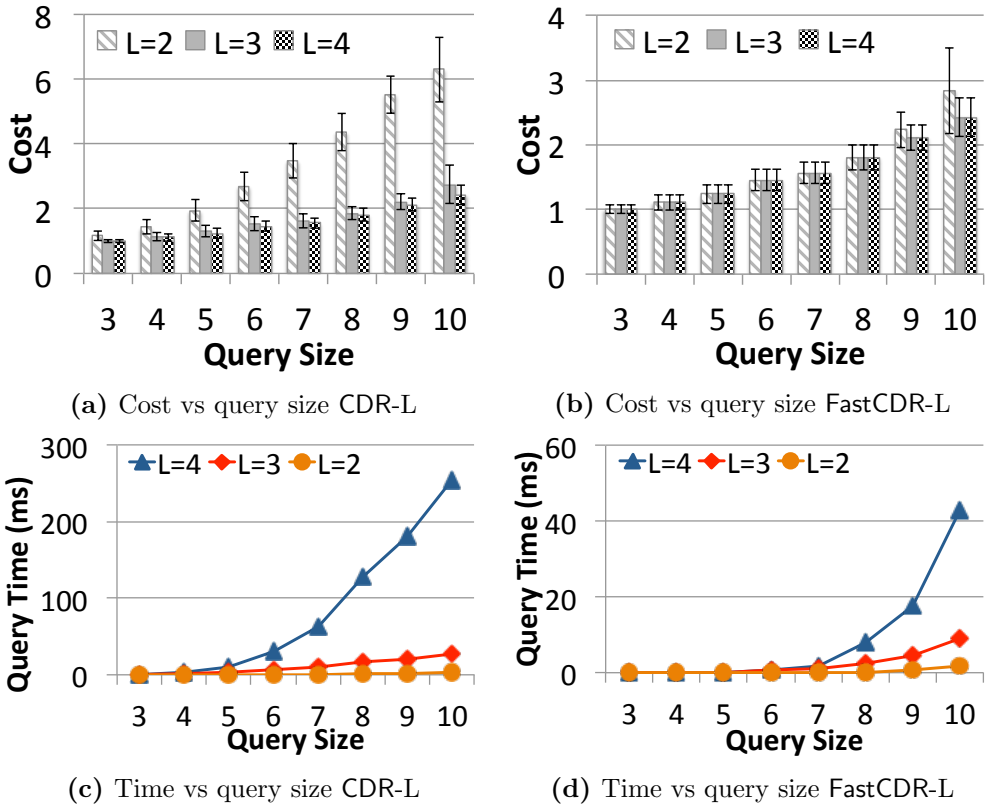
and FastCDR are constant in the cardinality value. Moreover, as depicted in Figure 3.16a, the quality of the results produced by FastCDR is not affected by the cardinality value.

3.7.8 Calibrating (Fast)CDR

Recall that the (Fast)CDR-L algorithms start by computing all the nodes of the relaxation tree for the first L levels (see Section 3.5.4), where L is a parameter.

Figure 3.17a and 3.17b show the impact of L on the cost (the values have been normalized using as a base the cost of (Fast)CDR-4 for query size 3). We notice that for $L = 2$ the CDR behaves reasonably well only for very small query sizes, while FastCDR produces qualitative relaxations up to query size 8. This behavior is expected, since for big query sizes the algorithm is trying to approximate the node cost distributions and then makes decisions based on too little information. Increasing L always improves cost. $L = 3$ results in a considerable improvement in cost, but the results show that further increases have negligible additional returns. We also compare the time performance in Figures 3.17c,3.17d. The results show that (Fast)CDR-4 quickly becomes expensive in terms of time. We conclude that using $L = 3$, CDR and FastCDR achieve the desirable trade-off between effectiveness and efficiency.

We also conduct experiments in Figure 3.18b with varying the number of the FastCDR histogram buckets between $5 - 40$, which has a negligible impact on time performance. We experience in Figure 3.18a that with more than 20 buckets the effect on the quality is minimal. The algorithm is also stable with respect to the data used.



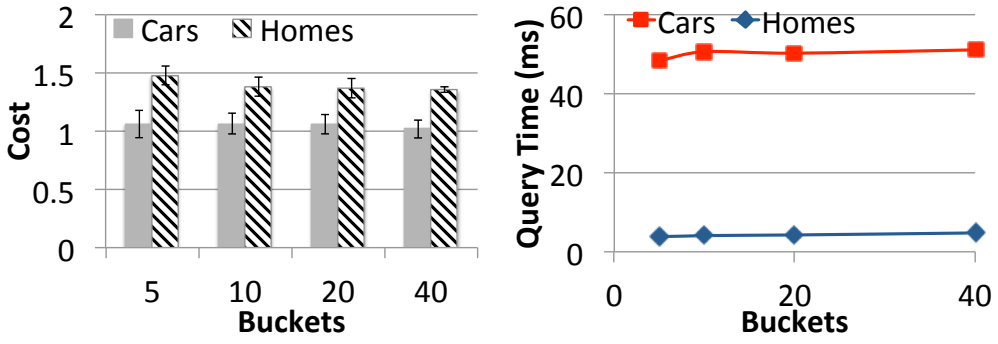
(a) Cost vs query size CDR-L

(b) Cost vs query size FastCDR-L

(c) Time vs query size CDR-L

(d) Time vs query size FastCDR-L

Figure 3.17: CDR and FastCDR behavior for different values of L at increasing query size.



(a) Cost vs number of buckets

(b) Time vs number of buckets

Figure 3.18: CDR behavior with increasing number of buckets in Homes and Cars datasets.

Chapter 4

Graph Query Reformulation

In the previous chapter we have studied the empty-answer problem, in which a user provides an over specified query for which no results exist in the database. In this chapter we tackle the complementary *information overload* problem. The user in this case is struggled with a very large number of results obtained from a too generic query. This causes an information overload that is as problematic as an empty-answer, for it is impossible to find the required object. We study this problem on structured databases and, more specifically, when the data is represented as a set of small graphs (graph database). This chapter describes our findings and solutions for this problem. We demonstrate that the problem of finding reformulations in graph databases is hard and we propose a solution with quality guarantee. We conclude with a thorough experimental assessment on several real and synthetic graph databases.

4.1 Contributions

Our main contributions are summarized as follows.

- We initiate the investigation of the problem of query reformulation in graph databases. We do so by formalizing the problem of finding a set of k reformulations of the input query that maximizes a linear combination of coverage and diversity.

- We show that our problem is **NP**-hard, as well as that our coverage function is monotone and submodular, and that our diversity measure is a pseudometric. These two properties allow us to adopt a greedy algorithm with provable quality guarantees.
- In order to guarantee efficiency, we devise a fast yet *exact* algorithm for the computation of the reformulation leading to the maximum objective-function increment.
- We perform an extensive evaluation on real and synthetic graph databases. Results confirm high quality and efficiency achieved by our method.

4.2 Outline

We first introduce the problem with an example in Section 4.3. Section 4.4 introduces the graph query reformulation problem and the notation used throughout the algorithmic solution in Section 4.5. In Section 4.6 we propose the experimental results with insights of time and quality of our algorithms.

4.3 Motivating example

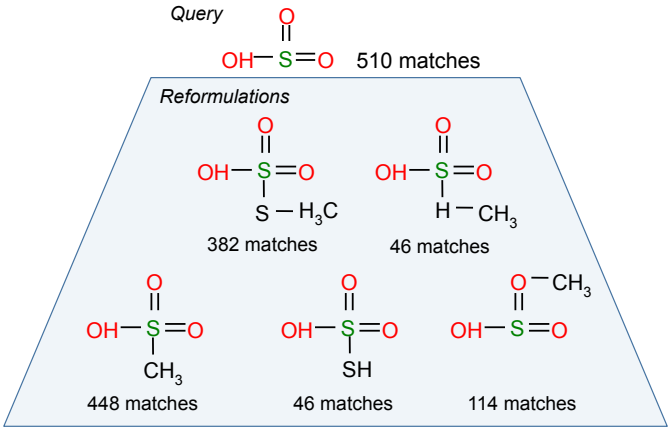


Figure 4.1: A graph query and a set of five reformulations produced by our method on the real-world database `AIDS`.

In the example in Figure 4.1 the data analyst is searching for `sulfonic acids`[1] in the database.

[1] `http://en.wikipedia.org/wiki/Sulfonic_acid`

Sulfonic acids are made of a `sulfonyl hydroxide` group (corresponding to the query in Figure 4.1) associated with some organic compound. Instead of directly searching independently for the various sulfonic acids, the data analyst issues the sulfonyl hydroxide group as a query, for which the database returns 510 graphs. In order to help the analyst in the exploratory search, the database proposes several reformulations of the original query with the associated number of matching graphs. This immediately provides a high-level summary of what the database contains and guidance for the analyst in further inspecting more specific queries.

4.4 Background and Problem

We now introduce a slightly different model from the one in Chapter 3, although we preserve the same notation for database and query we highlight the differences. The model we study here is a set of small graphs, called *graph database*.

4.4.1 Background

Let $\mathcal{D}$ be a *graph database* defined over a set of labels $\mathcal{L}$. Each element of $\mathcal{D}$ is a labeled graph $G : \langle N, E\rangle$, where N is a set of vertices, $E \subseteq N \times N$ is a set of edges, for which it exists a labeling function $\ell : N \cup E \to \mathcal{L}$ that assigns a label from $\mathcal{L}$ to each vertex in N and each edge in E. For presentation clarity, we assume the graphs in $\mathcal{D}$ to be undirected; however, all our methods can straightforwardly handle directed graphs as well without any significant modifications.

A *graph isomorphism* between two graphs $G_1 : \langle N_1, E_1\rangle$ with labeling function ℓ_1 and $G_2 : \langle N_2, E_2\rangle$ with labeling function ℓ_2, is a bijective function $\mu : N_1 \to N_2$ such that: (i) $\ell_1(u) = \ell_2(\mu(u))$, for each $u \in N_1$, and (ii) for every $(u, v) \in E_1$, $(\mu(u), \mu(v)) \in E_2$ and $\ell_1(u, v) = \ell_2(\mu(u), \mu(v))$. If a graph isomorphism exists between G_1 and G_2, we say that G_1 and G_2 are *isomorphic*. If a graph isomorphism exists between G_1 and a subgraph of G_2, we say that G_1 is *subgraph isomorphic* to G_2, and we denote it by $G_1 \sqsubseteq G_2$.

A *query* Q to a graph database $\mathcal{D}$ is a *connected* labeled graph. The answer to Q is the set $\mathcal{D}_Q = \{G \in \mathcal{D} \mid Q \sqsubseteq G\}$ of all graphs in $\mathcal{D}$, which Q is subgraph isomorphic to. We refer to $\mathcal{D}_Q$ as the *results* or the *result set* of Q.

Given a query Q, a *reformulated query* (or, simply, a *reformulation*) Q' of Q is a connected labeled graph such that $Q \sqsubseteq Q'$. We denote by $\mathbb{S}_Q$ the set of all possible reformulations of Q in the entire graph database $\mathcal{D}$, i.e., $\mathbb{S}_Q = \bigcup_{G \in \mathcal{D}} \{Q' \mid Q \sqsubseteq Q' \sqsubseteq G,\ Q' \text{connected}\}$. It can be observed that, by definition, the answer to every reformulated query Q' is always a subset of the answer to the original query Q, i.e., $\mathcal{D}_{Q'} \subseteq \mathcal{D}_Q$.

4.4.2 Problem Definition

Given a graph database $\mathcal{D}$ and a query Q, our goal is to find a set of k reformulations of Q that captures well the results $\mathcal{D}_Q$ of Q. More specifically, we aim at selecting a set of reformulations of cardinality k that exhibits (i) high *coverage* of the result set $\mathcal{D}_Q$, and (ii) high *diversity* among the subsets of $\mathcal{D}_Q$ identified by the single reformulations. We next formalize these concepts.

The coverage of a set of reformulations $\mathcal{Q}$ is defined as the number of results in $\mathcal{D}_Q$ captured by the reformulations:

$$cov(\mathcal{Q}) = \left| \bigcup_{Q' \in \mathcal{Q}} \mathcal{D}_{Q'} \right|, \tag{4.1}$$

while the diversity between two queries Q' and Q'' is defined as the number of uncommon results:

$$\begin{aligned} div(Q', Q'') &= \left| \mathcal{D}_{Q'} \cup \mathcal{D}_{Q''} \right| - \left| \mathcal{D}_{Q'} \cap \mathcal{D}_{Q''} \right| = \\ &= \left| \mathcal{D}_{Q'} \right| + \left| \mathcal{D}_{Q''} \right| - 2 \left| \mathcal{D}_{Q'} \cap \mathcal{D}_{Q''} \right|. \end{aligned} \tag{4.2}$$

Overall, the function we aim at maximizing is:

$$f(\mathcal{Q}) = cov(\mathcal{Q}) + \lambda \sum_{Q', Q'' \in \mathcal{Q}} div(Q', Q''), \tag{4.3}$$

where $\lambda \in [0, 1]$ is a parameter that trades off between coverage and diversity.[2] Finally, the problem we tackle in this work is formally defined as follows:

[2] λ plays also the role of a regularization factor for the (possibly) different scales of the two terms of f.

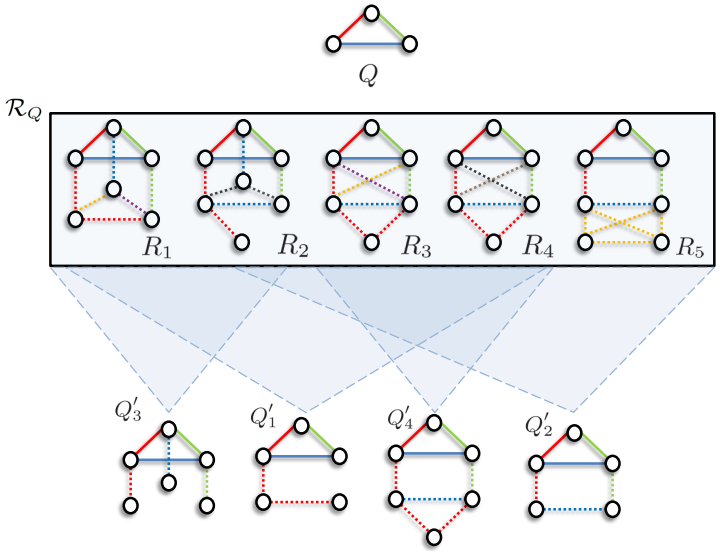


Figure 4.2: The Graph Query Reformulation problem.

Problem 1 (Graph Query Reformulation)**.** *Given a graph database $\mathcal{D}$, a query Q, and an integer k, find a set $\mathcal{Q}^*$ of reformulations of Q such that:*

$$\mathcal{Q}^* = \underset{\mathcal{Q} \subseteq \mathbb{S}_Q}{\text{argmax}} \quad f(\mathcal{Q})$$
$$\textit{subject to} \qquad |\mathcal{Q}| = k.$$

For $\lambda = 0$, Problem 1 corresponds to the well-known MaximumCoverage problem [H+97], which is known to be **NP**-hard. As a result, Problem 1 is **NP**-hard as well.

Example 4.1. *Figure 4.2 shows an example of the* Graph Query Reformulation *problem. The figure depicts a query Q, its corresponding result set $\mathcal{D}_Q = \{R_1, \ldots, R_5\}$, and four reformulations $Q'_1, \ldots, Q'_4$. The subset of $\mathcal{D}_Q$ captured by each reformulation is as follows: the first four results and the last four results form the result set of Q'_1 and Q'_2, respectively, while the results of Q'_3 and Q'_4 are $\{R_1, R_2\}$ and $\{R_3, R_4\}$, respectively. We also assume $\lambda = 0.3$.*

For $k = 2$, it can intuitively be observed that $\{Q'_3, Q'_4\}$ detect two of the main discriminating patterns arising from the result set $\mathcal{D}_Q$. The solution $\{Q'_1, Q'_2\}$ instead does not summarize $\mathcal{D}_Q$ equally well, as Q'_1 and Q'_2 identify two very general and not really informative patterns that are similar to one another and, as such, are both present in most of the results in $\mathcal{D}_Q$. This observation is acknowledged by the notions of coverage and diversity: $\{Q'_1, Q'_2\}$ have indeed slightly larger coverage than $\{Q'_3, Q'_4\}$, but they also

exhibit much smaller diversity, which makes the latter solution preferable. Hence, this example shows that coverage and diversity are both critical in order to find a valid set of reformulations. Our objective function f captures this main finding: $f(\{Q'_3, Q'_4\})$ is larger than $f(\{Q'_1, Q'_2\})$, thus $\{Q'_3, Q'_4\}$ is preferred to $\{Q'_1, Q'_2\}$ according to f.

4.5 Algorithmic Solution

We next focus on how to solve Problem 1. We first discuss a naïve solution based on frequent subgraph mining (Section 4.5.1). Then, we shift the attention to the proposed approach: we prove some properties of our objective function f (Section 4.5.2) , based on which we present a greedy algorithm exhibiting a $\frac{1}{2}$-approximation guarantee, while in Section 4.5.3 we treat the subproblem of finding the reformulation that maximizes the *marginal potential gain* (defined next), which is the key step of the greedy algorithm.

4.5.1 A naïve approach

The objective function defined in Equation (4.3) may resemble a notion of frequency: the more a reformulation covers a result set, the more frequent that reformulation is among the graphs in the result set.

This observation allows for defining a simple heuristic strategy to attack Problem 1, which is based on the well-known problem of *frequent subgraph mining* [YH02, HWPY04, NK04]: given a graph database and a threshold σ, find all subgraphs that are contained into at least σ graphs of the database. In our context we do not have a threshold-based problem definition; however, a natural yet straightforward way of adapting the frequent-subgraph-mining problem to our context exists, and it corresponds to ask for the top-k frequent subgraphs that are supergraphs of the input query graph.

The advantage of using this approach as a solution to Problem 1 is that the literature on frequent subgraph mining can be reused almost as is, since the adaptation of existing frequent-subgraph-mining techniques to this variant of the problem is trivial. Unfortunately, this simple approach is not guaranteed to produce high-quality results. Indeed, while the notion of frequency is related to the notion of coverage, the notion of diversity

Algorithm 6 Greedy

Input: A graph database $\mathcal{D}$, a query Q, an integer k
Output: A set of reformulated queries $\mathcal{Q}$
1: $\mathcal{Q} \leftarrow \emptyset$
2: **while** $|\mathcal{Q}| < k$ **do**
3: $\quad Q^* \leftarrow \arg\max_{Q' \in \mathbb{S}_Q \setminus \mathcal{Q}} \Delta_f(\mathcal{Q}, Q')$ ▷ Equation (4.4)
4: $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{Q^*\}$

is instead completely ignored. Taking into account diversity is crucial to find meaningful solutions, as clearly demonstrated in Example 4.1.

The poor effectiveness of this naïve strategy is also supported by experimental evidence (see Section 4.6).

4.5.2 An approach with quality guarantees

The algorithm we propose as a more principled solution to Problem 1 relies on a recent result by Borodin *et al.* [BLY12]. Given a universe of elements U, let $w : 2^U \rightarrow \mathbb{R}$ be a non-negative function measuring the quality of every subset of U, and $d : U \times U \rightarrow \mathbb{R}$ be a distance function between elements of U. Also, let g be a set function defined as a linear combination of w and the sum of pairwise distances computed according to d, i.e., for any $\hat{U} \subseteq U$, $g(\hat{U}) = w(\hat{U}) + \lambda \sum_{u',u'' \in \hat{U}} d(u', u'')$ (with $\lambda \in [0, 1]$ being a parameter). Finally, given a subset $U' \subset U$ and an element $u \notin U'$, let $\frac{1}{2}(w(U' \cup \{u\}) - w(U')) + \lambda(\sum_{u',u'' \in \hat{U}' \cup \{u\}} d(u', u'') - \sum_{u',u'' \in \hat{U}} d(u', u''))$ denote the *marginal potential gain* of the function g. Borodin *et al.* [BLY12] show that, for the problem of maximizing a set function like g under a cardinality constraint, a greedy algorithm that iteratively selects the element maximizing the marginal potential gain achieves a $\frac{1}{2}$-approximation guarantee if (i) the quality function w is monotone submodular, and (ii) the universe U spans a metric space, i.e., d is a metric. As shown in more detail next, this result remains true even if d is a *pseudometric*.

In our context, the universe U corresponds to the set $\mathbb{S}_Q$ of all possible reformulations of the input query Q, while the quality function corresponds to the coverage function *cov* and the distance between two elements is measured in terms of the diversity *div* between two reformulations. Also, given a set of reformulated queries $\mathcal{Q} \subseteq \mathbb{S}_Q$ and a reformulation $Q' \in \mathbb{S}_Q \setminus \mathcal{Q}$, and denoting by $\Delta_{cov}(\mathcal{Q}, Q') = cov(\mathcal{Q} \cup \{Q'\}) - cov(\mathcal{Q})$ the marginal gain of the coverage term and by $\Delta_{div}(\mathcal{Q}, Q') = \sum_{\hat{Q} \in \mathcal{Q}} div(\hat{Q}, Q')$ the marginal

gain of the diversity term, the marginal potential gain of our function f is defined as

$$\Delta_f(\mathcal{Q}, Q') = \frac{1}{2}\Delta_{cov}(\mathcal{Q}, Q') + \lambda\Delta_{div}(\mathcal{Q}, Q'). \tag{4.4}$$

Thus, to exploit the above result by Borodin *et al.* [BLY12] we need to prove that the function *cov* is monotone submodular, and *div* is a pseudometric. We next prove that these properties indeed hold.

Lemma 4.1. *$cov(\cdot)$ is a monotone submodular set function.*

Proof. A set function w is said monotone if for every $S \subseteq T$ it holds that $w(S) \leq w(T)$. It is straightforward to see from Equation (4.1) that the function *cov* possesses such a property. Concerning submodularity, a set function w is said submodular if for every $S \subseteq T$ and every $x \notin T$ it holds that $w(S \cup \{x\}) - w(S) \geq w(T \cup \{x\}) - w(T)$. Let $\mathcal{Q}'$ and $\mathcal{Q}''$, $\mathcal{Q}' \subseteq \mathcal{Q}''$, be two sets of reformulations and let $\hat{Q} \notin \mathcal{Q}''$ be a reformulation. It holds that:

$$\begin{aligned}
&\mathcal{Q}' \subseteq \mathcal{Q}'' \Rightarrow \\
&\quad \Rightarrow \textstyle\bigcup_{Q' \in \mathcal{Q}'} \mathcal{D}_{Q'} \subseteq \bigcup_{Q'' \in \mathcal{Q}''} \mathcal{D}_{Q''} \\
&\quad \Rightarrow \left|\mathcal{D}_{\hat{Q}} \cap \textstyle\bigcup_{Q' \in \mathcal{Q}'} \mathcal{D}_{Q'}\right| \leq \left|\mathcal{D}_{\hat{Q}} \cap \textstyle\bigcup_{Q'' \in \mathcal{Q}''} \mathcal{D}_{Q''}\right| \\
&\quad \Leftrightarrow \left|\mathcal{D}_{\hat{Q}}\right| - \left|\mathcal{D}_{\hat{Q}} \cap \textstyle\bigcup_{Q' \in \mathcal{Q}'} \mathcal{D}_{Q'}\right| \geq \left|\mathcal{D}_{\hat{Q}}\right| - \left|\mathcal{D}_{\hat{Q}} \cap \textstyle\bigcup_{Q'' \in \mathcal{Q}''} \mathcal{D}_{Q''}\right| \\
&\quad \Leftrightarrow cov(\mathcal{Q}' \cup \{\hat{Q}\}) - cov(\mathcal{Q}') \geq cov(\mathcal{Q}'' \cup \{\hat{Q}\}) - cov(\mathcal{Q}'').
\end{aligned}$$

Then, *cov* is submodular, and the lemma follows. □

Lemma 4.2. *$div(\cdot, \cdot)$ is a pseudometric.*

Proof. To be a pseudometric, *div* needs to satisfy three of the classic properties possessed by any metric, i.e., (i) non-negativity ($div(x, y) \geq 0$), (ii) symmetry ($div(x, y) = div(y, x)$), and (iii) triangle inequality, while, concerning (iv) reflexivity, it is sufficient that $div(x, y) = 0$ holds if $x = y$, while possibly $div(x, y) = 0$ for distinct objects $x \neq y$. It is easy to see that the (i), (ii), and (iv) hold by definition. Concerning triangle inequality, we need to prove that $div(x, y) + div(y, z) \geq div(x, z)$, which corresponds to show that $|\mathcal{D}_x \cap \mathcal{D}_y| + |\mathcal{D}_y \cap \mathcal{D}_z| - |\mathcal{D}_y| - |\mathcal{D}_x \cap \mathcal{D}_z| \leq 0$.

By the inclusion-exclusion principle, i.e., $|X| + |Y| = |X \cup Y| + |X \cap Y|$, for two sets X and Y, we can derive:

$$\begin{aligned}
&|\mathcal{D}_x \cap \mathcal{D}_y| + |\mathcal{D}_y \cap \mathcal{D}_z| - |\mathcal{D}_y| - |\mathcal{D}_x \cap \mathcal{D}_z| = \\
&\quad = |(\mathcal{D}_x \cap \mathcal{D}_y) \cup (\mathcal{D}_y \cap \mathcal{D}_z)| + |\mathcal{D}_x \cap \mathcal{D}_y \cap \mathcal{D}_z| - |\mathcal{D}_y| - |\mathcal{D}_x \cap \mathcal{D}_z| \\
&\quad = |(\mathcal{D}_y \cap (\mathcal{D}_x \cup \mathcal{D}_z)| + |\mathcal{D}_x \cap \mathcal{D}_y \cap \mathcal{D}_z| - |\mathcal{D}_y| - |\mathcal{D}_x \cap \mathcal{D}_z| \\
&\quad \leq |(\mathcal{D}_y \cap (\mathcal{D}_x \cup \mathcal{D}_z)| + |\mathcal{D}_x \cap \mathcal{D}_y \cap \mathcal{D}_z| - |\mathcal{D}_y| - |\mathcal{D}_x \cap \mathcal{D}_y \cap \mathcal{D}_z| \\
&\quad = |(\mathcal{D}_y \cap (\mathcal{D}_x \cup \mathcal{D}_z)| - |\mathcal{D}_y| \leq 0,
\end{aligned}$$

where the latter inequality holds as $|X \cap Y| \leq |X|$. The lemma follows. □

The proposed Greedy algorithm, whose pseudocode is shown as Algorithm 6, iteratively selects the reformulation Q^* that brings the maximum marginal potential gain to the objective function f, until k reformulations have been selected.

The following result holds.

Theorem 4.3. Greedy *is a* $\frac{1}{2}$*-approximation algorithm for the* Graph Query Reformulation *problem.*

Proof. Looking at the proof of Theorem 1 in [BLY12], it is easy to see that such a theorem remains true even in case of pseudometrics, as the reflexivity axiom is not exploited at all. The proof is completed by the results stated in Lemma 4.1 (monotonicity and submodularity) and Lemma 4.2 (pseudometric). □

4.5.3 Maximizing the marginal potential gain

The proposed Greedy needs to face two main challenges:

- Finding the element that maximizes the marginal potential gain is very difficult in our context, as it corresponds to selecting a reformulated query $Q^* \in \mathbb{S}_Q \setminus \mathcal{Q}$ that achieves the desired maximum objective-function increment, and, in the worst case, this requires to enumerate all possible subgraphs of each graph $G \in \mathcal{D}$ that Q is subgraph isomorphic to. Such a problem corresponds to counting all subgraph-isomorphic structures in a graph and is known to be #**P**-complete [Val79].

- Our function f is *non-monotone*: this makes the design of pruning strategies non-trivial at all (e.g., traditional downward-closure-based algorithms do not work).

In the following we present the proposed solution to the most critical step of the Greedy algorithm of finding the element that maximizes the marginal potential gain. We aim at solving this step efficiently while still guaranteeing optimality, as this is needed to preserve the approximation guarantee of Theorem 4.3. To this end, we devise an algorithm, called Fast_MMPG, that visits the search space in a smart way based on two main findings: (i) an efficient computation of the marginal potential gain Δ_f, and (ii) an upper bound on the maximum marginal potential gain achievable by a set of reformulations that is used to prune significant portions of the search space. We next report the details of each of these findings and describe the proposed Fast_MMPG.

Computing Δ_f in linear time

We show here how to efficiently solve the frequently-occurring subproblem of computing Δ_f when a reformulation Q' is given. According to Equation (4.4), Δ_f is a linear combination of Δ_{cov} and Δ_{div}. The marginal gain Δ_{cov} can be computed by a simple scan of the results in $\mathcal{D}_{Q'}$, thus taking $\mathcal{O}(|\mathcal{D}_{Q'}|)$ time. As far as Δ_{div} concerns, a naïve computation would instead consider the results of all reformulations in the current set $\mathcal{Q}$, thus requiring quadratic time. We next show how to carry out the computation in a smarter way, so as to take $\mathcal{O}(|\mathcal{D}_{Q'}|)$ time as well. The idea is to adopt a vector that keeps track of how many reformulations (among the ones already computed) capture any single result of the input query. This simple structure is sufficient to overcome the quadratic cost of the pairwise comparison in Equation (4.3).

Given a query Q, let $\{R_1, \ldots, R_n\}$ denote the graphs in its result set $\mathcal{D}_Q$. As stated above, the results of every reformulated query Q' of Q are guaranteed to be a subset of $\mathcal{D}_Q$. Therefore, one can alternatively keep track of the results of Q' by using a binary n-dimensional vector $\mathbf{x}_{Q'}$, where $\mathbf{x}_{Q'}[i] = 1$ if and only if $R_i \in \mathcal{D}_{Q'}$. Given a set of reformulations $\mathcal{Q} \subseteq \mathbb{S}_Q$, let also $\mathbf{m}_\mathcal{Q} = \sum_{\hat{Q} \in \mathcal{Q}} \mathbf{x}_{\hat{Q}}$ be an n-dimensional integer vector whose i-th entry contains the number of reformulations in $\mathcal{Q}$ having R_i among their results. We hereinafter refer to $\mathbf{m}_\mathcal{Q}$ as the *multiplicity vector*. We also use $\|\mathbf{v}\|$ to denote the L_1-norm of a vector $\mathbf{v}$, i.e., the sum of the elements in $\mathbf{v}$. In the next theorem

we show how to exploit $\mathbf{m}_{\mathcal{Q}}$ and $\mathbf{x}_{Q'}$ to achieve the desired $\mathcal{O}(|\mathcal{D}_{Q'}|)$-time computation of Δ_{div}.

Theorem 4.4. *Given a set of reformulations $\mathcal{Q} \subseteq \mathbb{S}_Q$ and a reformulation $Q' \in \mathbb{S}_Q \setminus \mathcal{Q}$, the marginal gain $\Delta_{div}(\mathcal{Q}, Q')$ is equal to:*

$$\Delta_{div}(\mathcal{Q}, Q') = \|\mathbf{m}_{\mathcal{Q}}\| + |\mathcal{Q}| \times |\mathcal{D}_{Q'}| - 2\ \mathbf{m}_{\mathcal{Q}} \cdot \mathbf{x}_{Q'}.$$

Proof. $\Delta_{div}(\mathcal{Q}, Q')$ is equal to:

$$\begin{aligned}
\Delta_{div}(\mathcal{Q}, Q') &= \sum_{\hat{Q} \in \mathcal{Q}} div(\hat{Q}, Q') = \\
&= \sum_{\hat{Q} \in \mathcal{Q}} \Big(|\mathcal{D}_{\hat{Q}}| + |\mathcal{D}_{Q'}| - 2|\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q'}| \Big) \\
&= \underbrace{\sum_{\hat{Q} \in \mathcal{Q}} |\mathcal{D}_{\hat{Q}}|}_{\|\mathbf{m}_{\mathcal{Q}}\|} + \underbrace{\sum_{\hat{Q} \in \mathcal{Q}} |\mathcal{D}_{Q'}|}_{|\mathcal{Q}| \times |\mathcal{D}_{Q'}|} - 2 \sum_{\hat{Q} \in \mathcal{Q}} |\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q'}|. \qquad (4.5)
\end{aligned}$$

By noticing that the i-th entry of the vector $\mathbf{m}_{\mathcal{Q}}$ can alternatively be expressed as $\mathbf{m}_{\mathcal{Q}}[i] = \sum_{\hat{Q} \in \mathcal{Q}} \mathbb{1}[R_i \in \mathcal{D}_{\hat{Q}}]$ (where $\mathbb{1}[\cdot]$ is the indicator function), the term $\sum_{\hat{Q} \in \mathcal{Q}} |\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q'}|$ in (4.5) can be rewritten as:

$$\begin{aligned}
\sum_{\hat{Q} \in \mathcal{Q}} |\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q'}| &= \sum_{\hat{Q} \in \mathcal{Q}} \sum_{\substack{i \in [1..n], \\ R_i \in \mathcal{D}_{Q'}}} \mathbb{1}[R_i \in \mathcal{D}_{\hat{Q}}] \\
&= \sum_{\substack{i \in [1..n], \\ R_i \in \mathcal{D}_{Q'}}} \underbrace{\sum_{\hat{Q} \in \mathcal{Q}} \mathbb{1}[R_i \in \mathcal{D}_{\hat{Q}}]}_{\mathbf{m}_{\mathcal{Q}}[i]} = \\
&= \sum_{i \in [1..n]} \big(\mathbf{x}_{Q'}[i] \times \mathbf{m}_{\mathcal{Q}}[i] \big) = \ \mathbf{m}_{\mathcal{Q}} \cdot \mathbf{x}_{Q'}. \qquad (4.6)
\end{aligned}$$

The proof is completed combining (4.5) and (4.6). □

It is easy to see that the rewriting in Theorem 4.4 allows for computing Δ_{div} linearly in $|\mathcal{D}_{Q'}|$ as the terms $\|\mathbf{m}_{\mathcal{Q}}\|$ and $|\mathcal{Q}| \times |\mathcal{D}_{Q'}|$ are constant, while the scalar product $\mathbf{m}_{\mathcal{Q}} \cdot \mathbf{x}_{Q'}$ requires a scan of only the $|\mathcal{D}_{Q'}|$ non-zero entries of $\mathbf{x}_{Q'}$.

Upper bound on Δ_f

Here we derive an upper bound on the marginal potential gain Δ_f (Equation (4.4)) exhibited by a set of reformulations. Specifically, given a reformulated query Q', let $\mathcal{T}_{Q'}$ denote the set of all reformulations that Q' is subgraph isomorphic to, i.e., $\mathcal{T}_{Q'} = \{Q'' \in \mathbb{S}_Q \setminus \mathcal{Q} \mid Q' \sqsubseteq Q''\}$. We show how to bound the maximum marginal potential gain achievable by a reformulation in $\mathcal{T}_{Q'}$ in a very efficient manner, that is by looking only at the reformulation Q'. The ultimate goal is to exploit the resulting upper bound to early recognize (and skip) unnecessary portions of the search space.

We derive our upper bound by studying which results in $\mathcal{D}_Q$ should be captured by a reformulation in $\mathcal{T}_{Q'}$ to achieve maximum marginal potential gain. To this end, let $\mathcal{D}_Q = \{R_1, \ldots, R_n\}$ denote the results of the original query Q. We assume that a set of reformulations $\mathcal{Q} \subseteq \mathbb{S}_Q$ have been computed and kept track of the results identified by $\mathcal{Q}$ by the multiplicity vector $\mathbf{m}_\mathcal{Q}$ introduced above. Let $Q_1, Q_2 \in \mathbb{S}_Q \setminus \mathcal{Q}$ be two reformulations such that their corresponding result sets differ by only one element, i.e., $\mathcal{D}_{Q_2} = \mathcal{D}_{Q_1} \cup \{R_j\}$. We want to study when the marginal potential gain brought by Q_2 is no less than the one given by Q_1, i.e., when $\phi_f = \Delta_f(\mathcal{Q}, Q_2) - \Delta_f(\mathcal{Q}, Q_1) \geq 0$. Note that we focus on a pair of reformulations whose result sets differ by only one element in order to simplify the presentation of the theoretical results therein. This however does not result in any loss of generality, as shown later in Theorem 4.8.

We start our reasoning by showing how to profitably rearrange the quantities $\Delta_{cov}(\mathcal{Q}, Q_2) - \Delta_{cov}(\mathcal{Q}, Q_1)$ (Lemma 4.5) and $\Delta_{div}(\mathcal{Q}, Q_2) - \Delta_{div}(\mathcal{Q}, Q_1)$ (Lemma 4.6).

Lemma 4.5. *Let $Q_1, Q_2 \in \mathbb{S}_Q \setminus \mathcal{Q}$, s.t. $\mathcal{D}_{Q_2} = \mathcal{D}_{Q_1} \cup \{R_j\}$. It holds that $\Delta_{cov}(\mathcal{Q}, Q_2) - \Delta_{cov}(\mathcal{Q}, Q_1) = \mathbb{1}[\mathbf{m}_\mathcal{Q}[j] = 0]$.*

Proof.

$$
\begin{aligned}
&\Delta_{cov}(\mathcal{Q}, Q_2) - \Delta_{cov}(\mathcal{Q}, Q_1) = \\
&= \sum_{\substack{i \in [1..n], \\ R_i \in \mathcal{D}_{Q_2}}} \mathbb{1}[\mathbf{m}_\mathcal{Q}[i] = 0] \; - \sum_{\substack{i \in [1..n], \\ R_i \in \mathcal{D}_{Q_1}}} \mathbb{1}[\mathbf{m}_\mathcal{Q}[i] = 0] \\
&= \sum_{\substack{i \in [1..n], \\ R_i \in \mathcal{D}_{Q_1}}} \mathbb{1}[\mathbf{m}_\mathcal{Q}[i] = 0] + \mathbb{1}[\mathbf{m}_\mathcal{Q}[j] = 0] \; - \sum_{\substack{i \in [1..n], \\ R_i \in \mathcal{D}_{Q_1}}} \mathbb{1}[\mathbf{m}_\mathcal{Q}[i] = 0] \\
&= \mathbb{1}[\mathbf{m}_\mathcal{Q}[j] = 0].
\end{aligned}
$$

□

Lemma 4.6. *It holds that* $\Delta_{div}(\mathcal{Q}, Q_2) - \Delta_{div}(\mathcal{Q}, Q_1) = |\mathcal{Q}| - 2\mathbf{m}_{\mathcal{Q}}[j]$.

Proof.

$$\begin{aligned}
&\Delta_{div}(\mathcal{Q}, Q_2) - \Delta_{div}(\mathcal{Q}, Q_1) = \\
&= \sum_{\hat{Q}\in\mathcal{Q}} div(\hat{Q}, Q_2) - \sum_{\hat{Q}\in\mathcal{Q}} div(\hat{Q}, Q_1) \\
&= \sum_{\hat{Q}\in\mathcal{Q}} \Big(|\mathcal{D}_{\hat{Q}}| + \underbrace{|\mathcal{D}_{Q_2}|}_{|\mathcal{D}_{Q_1}|+1} - 2|\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q_2}|\Big) \\
&\qquad - \sum_{\hat{Q}\in\mathcal{Q}} \Big(|\mathcal{D}_{\hat{Q}}| + |\mathcal{D}_{Q_1}| - 2|\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q_1}|\Big) \\
&= |\mathcal{Q}| - 2\sum_{\hat{Q}\in\mathcal{Q}} \Big(|\mathcal{D}_{\hat{Q}} \cap (\mathcal{D}_{Q_1} \cup \{R_j\})| - |\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q_1}|\Big) \\
&= |\mathcal{Q}| - 2\sum_{\hat{Q}\in\mathcal{Q}} \Big(|\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q_1}| + \mathbb{1}[R_j \in \mathcal{D}_{\hat{Q}}] - |\mathcal{D}_{\hat{Q}} \cap \mathcal{D}_{Q_1}|\Big) \\
&= |\mathcal{Q}| - 2\underbrace{\sum_{\hat{Q}\in\mathcal{Q}} \mathbb{1}[R_j \in \mathcal{D}_{\hat{Q}}]}_{\mathbf{m}_{\mathcal{Q}}[j]} = |\mathcal{Q}| - 2\mathbf{m}_{\mathcal{Q}}[j].
\end{aligned}$$

□

We now exploit Lemma 4.5 and 4.6 to show the desired condition about $\phi_f = \Delta_f(\mathcal{Q}, Q_2) - \Delta_f(\mathcal{Q}, Q_1) \geq 0$. We formally state this in the next lemma.

Lemma 4.7. $\phi_f \geq 0$ *if and only if* $\mathbf{m}_{\mathcal{Q}}[j] \leq \frac{1}{2}|\mathcal{Q}|$.

Proof. By Lemma 4.5 and 4.6 it results that:

$$\begin{aligned}
\phi_f = \Delta_f(\mathcal{Q}, Q_2) - \Delta_f(\mathcal{Q}, Q_1) = \\
&= \frac{1}{2}\left(\Delta_{cov}(\mathcal{Q}, Q_2) - \Delta_{cov}(\mathcal{Q}, Q_1)\right) + \lambda\left(\Delta_{div}(\mathcal{Q}, Q_2) - \Delta_{div}(\mathcal{Q}, Q_1)\right) \\
&= \frac{1}{2}\mathbb{1}[\mathbf{m}_{\mathcal{Q}}[j] = 0] + \lambda(|\mathcal{Q}| - 2\mathbf{m}_{\mathcal{Q}}[j]).
\end{aligned}$$

From the latter equality, it can be observed that, if $\mathbf{m}_{\mathcal{Q}}[j] = 0$, then $\phi_f = \frac{1}{2} + \lambda|\mathcal{Q}| > 0$. Otherwise, if $\mathbf{m}_{\mathcal{Q}}[j] > 0$, then $\phi_f = \lambda(|\mathcal{Q}| - 2\mathbf{m}_{\mathcal{Q}}[j])$, which is ≥ 0 when $\mathbf{m}_{\mathcal{Q}}[j] \leq \frac{1}{2}|\mathcal{Q}|$. The lemma follows. □

Lemma 4.7 shows a condition about which results are "worth" to be captured by a reformulation in order to achieve maximum marginal potential gain: ideally, the best reformulation Q^* should contain in its result set $\mathcal{D}_{Q^*}$ *all and only* those results R_j whose corresponding multiplicity $\mathbf{m}_\mathcal{Q}[j]$ is no more than $\frac{1}{2}|\mathcal{Q}|$. To be precise, actually, the results R_j such that $\mathbf{m}_\mathcal{Q}[j] = \frac{1}{2}|\mathcal{Q}|$ do not affect optimality: they can either be present in $\mathcal{D}_{Q^*}$ or not.

We exploit the above reasoning to derive our upper bound on the maximum marginal potential gain exhibited by a reformulation in $\mathcal{T}_{Q'}$. We denote such an upper bound by $\overline{\Delta_f}(\mathcal{Q}, Q')$ and we formally state it in the next Theorem 4.8. Particularly, we express $\overline{\Delta_f}(\mathcal{Q}, Q')$ in terms of three n-dimensional binary vectors: $\mathbf{u}_\mathcal{Q}$ and $\mathbf{v}_\mathcal{Q}$, which keep track of the results in $\mathcal{D}_Q$ exhibiting null multiplicity and multiplicity no more than $\frac{1}{2}|\mathcal{Q}|$, respectively ($\mathbf{u}_\mathcal{Q}[i] = 1$ if and only if $\mathbf{m}_\mathcal{Q}[i] = 0$, and $\mathbf{v}_\mathcal{Q}[i] = 1$ if and only if $0 < \mathbf{m}_\mathcal{Q}[i] \leq \frac{1}{2}|\mathcal{Q}|$), and $\mathbf{x}_{Q^*}$, which keeps track of the results that the best reformulation Q^* should ideally capture and is defined as the Hadamard (i.e., element-wise) product between $\mathbf{v}_\mathcal{Q}$ and $\mathbf{x}_{Q'}$, i.e., $\mathbf{x}_{Q^*} = \mathbf{v}_\mathcal{Q} \circ \mathbf{x}_{Q'}$. The value of $\overline{\Delta_f}(\mathcal{Q}, Q')$ is as follows.

Theorem 4.8. *For a reformulation $Q' \in \mathbb{S}_Q \setminus \mathcal{Q}$ it holds that*

$$\max_{Q'' \in \mathcal{T}_{Q'}} \Delta_f(\mathcal{Q}, Q'') \;\leq\; \overline{\Delta_f}(\mathcal{Q}, Q') \;= \\ = \frac{1}{2}\mathbf{u}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} + \lambda \left(\|\mathbf{m}_\mathcal{Q}\| + |\mathcal{Q}| \times \|\mathbf{x}_{Q^*}\| - 2\mathbf{m}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} \right),$$

Proof. By Lemma 4.7, we know how the result set of the best reformulation Q^* should be: the content of this best result set $\mathcal{D}_{Q^*}$ is expressed by the binary vector $\mathbf{x}_{Q^*} = \mathbf{v}_\mathcal{Q} \circ \mathbf{x}_{Q'}$. Based on this, Equation (4.4) can be rewritten as follows:

$$\overline{\Delta_f}(\mathcal{Q}, Q') = \Delta_f(\mathcal{Q}, Q^*) = \\ = \frac{1}{2}\Delta_{cov}(\mathcal{Q}, Q^*) + \lambda \Delta_{div}(\mathcal{Q}, Q^*) \\ = \frac{1}{2}\mathbf{u}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} + \lambda \left(\|\mathbf{m}_\mathcal{Q}\| + |\mathcal{Q}| \times |\mathcal{D}_{Q^*}| - 2\mathbf{m}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} \right) \\ = \frac{1}{2}\mathbf{u}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} + \lambda \left(\|\mathbf{m}_\mathcal{Q}\| + |\mathcal{Q}| \times \|\mathbf{x}_{Q^*}\| - 2\mathbf{m}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} \right),$$

where the third equality above derives from Theorem 4.4. □

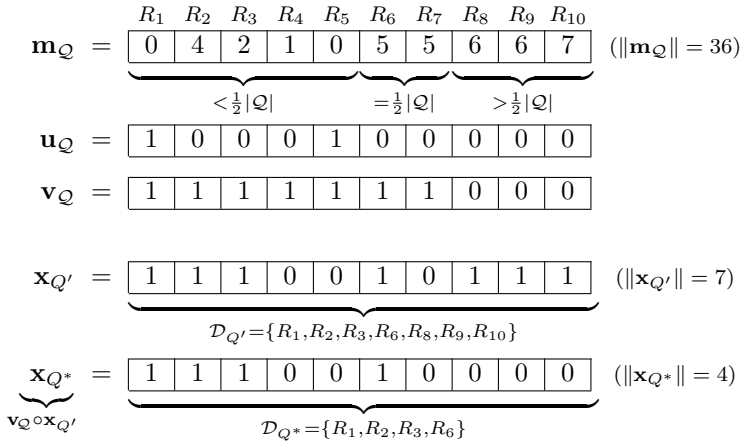


Figure 4.3: Illustration of the computation of the upper bound stated in Theorem 4.8.

The computation of the upper bound $\overline{\Delta_f}(\mathcal{Q}, Q')$ is really fast: it takes $\mathcal{O}(|\mathcal{D}_{Q'}|)$ time, as all the terms of the expression in Theorem 4.8 either are constant or can be computed by considering the $|\mathcal{D}_{Q'}|$ non-zero entries of the vector $\mathbf{x}_{Q'}$.

Example 4.2. *Figure 4.3 shows an example of the computation of the upper bound* $\overline{\Delta_f}$*. We consider an input query* Q *whose results are* $\mathcal{D}_Q = \{R_1, \ldots, R_{10}\}$*. We also assume that a set* $\mathcal{Q}$ *of* $|\mathcal{Q}| = 10$ *reformulations have already been computed and that* $\lambda = 0.5$*. The integer vector* $\mathbf{m}_\mathcal{Q}$ *contains the multiplicity of the results in* $\mathcal{Q}$*, i.e., each entry* $\mathbf{m}_\mathcal{Q}[i]$ *contains the number of reformulations in* $\mathcal{Q}$ *having* R_i *among their results, while the binary vectors* $\mathbf{u}_\mathcal{Q}$ *and* $\mathbf{v}_\mathcal{Q}$ *keep track of the results in* $\mathcal{D}_Q$ *having null multiplicity and multiplicity* $\leq \frac{1}{2}|\mathcal{Q}|$*, respectively. For a given reformulation* Q'*, the binary vectors* $\mathbf{x}_{Q'}$ *and* $\mathbf{x}_{Q^*}$ *denote the actual result set* $R_{Q'}$ *and the result set of the* ideal *reformulation* Q^* *that can be generated from* Q'*, respectively. According to Lemma 4.7, the result set of* Q^* *is given by all and only the results of* Q' *whose multiplicity is* $\leq \frac{1}{2}|\mathcal{Q}|$*, that is* $\mathbf{x}_{Q^*} = \mathbf{v}_\mathcal{Q} \circ \mathbf{x}_{Q'}$*.*

The marginal gain of the coverage term $\Delta_{cov}(\mathcal{Q}, Q^*)$ *is equal to the scalar product* $\mathbf{u}_\mathcal{Q} \cdot \mathbf{x}_{Q^*}$*; therefore,* $\Delta_{cov}(\mathcal{Q}, Q^*) = 1$*. According to Theorem 4.4, the marginal gain of the diversity term is* $\Delta_{div}(\mathcal{Q}, Q^*) = \|\mathbf{m}_\mathcal{Q}\| + |\mathcal{Q}| \times |\mathbf{x}_{Q^*}| - 2\mathbf{m}_\mathcal{Q} \cdot \mathbf{x}_{Q^*} = 36 + 10 \times 4 - 2 \times 11 = 54$*. As a result, the upper bound of* Q' *is* $\overline{\Delta_f}(\mathcal{Q}, Q') = \Delta_f(\mathcal{Q}, Q^*) = \frac{1}{2}\Delta_{cov}(\mathcal{Q}, Q^*) + \lambda\Delta_{div}(\mathcal{Q}, Q^*) = 27.5$*, while the actual marginal potential gain of* Q' *is* $\Delta_f(\mathcal{Q}, Q') = \frac{1}{2}\Delta_{cov}(\mathcal{Q}, Q') + \lambda\Delta_{div}(\mathcal{Q}, Q') = \frac{1}{2}\mathbf{u}_\mathcal{Q} \cdot \mathbf{x}_{Q'} + \lambda(\|\mathbf{m}_\mathcal{Q}\| + |\mathcal{Q}| \times |\mathbf{x}_{Q'}| - 2\mathbf{m}_\mathcal{Q} \cdot \mathbf{x}_{Q'}) = 23.5$*.*

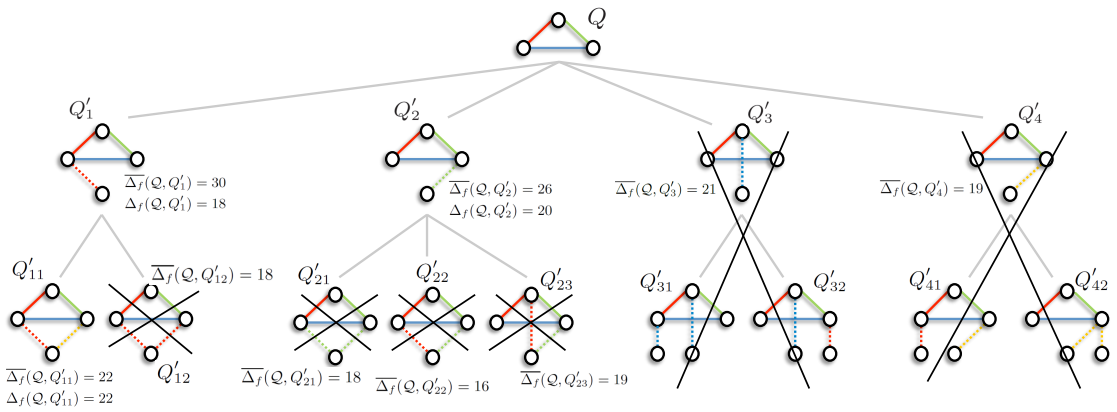


Figure 4.4: Illustration of the Fast_MMPG algorithm

The Fast_MMPG algorithm

We exploit the findings discussed above in order to efficiently find a reformulation exhibiting the maximum marginal potential gain. We assume our search space $\mathbb{S}_Q \setminus \mathcal{Q}$ organized as a tree $\mathcal{T}$, whose root corresponds to the input query Q, while the children of each node (reformulation) Q' correspond to all reformulations generated by adding a single edge to Q'. According to this definition, a reformulation can in principle have multiple fathers; this can however be avoided by borrowing standard mechanisms from frequent subgraph mining,[3] hence we can safely assume that each reformulation in $\mathcal{T}$ has actually a single father. As already anticipated, we also denote by $\mathcal{T}_{Q'}$ the subtree of $\mathcal{T}$ rooted at Q'.

The proposed Fast_MMPG algorithm, whose outline is reported as Algorithm 7, visits the various nodes Q' of the tree $\mathcal{T}$ in non-increasing ordering of their upper-bound $\overline{\Delta_f}(\mathcal{Q}, Q')$—the rationale here is that larger upper bounds correspond to more promising subtrees. To this end, a priority queue $\mathbf{P}$ is used. At the beginning, $\mathbf{P}$ contains the children of the original query Q (Line 2), and the algorithm processes the reformulations in $\mathbf{P}$ until it becomes empty or the maximum upper bound therein does not exceed the best-so-far value Δ_f^* (Lines 3–13). For each reformulation Q' extracted from $\mathbf{P}$, the algorithm first computes the marginal potential gain $\Delta_f(\mathcal{Q}, Q')$ (according to Theorem 4.4), and uses it to possibly update Δ_f^* (Lines 5–8). $\Delta_f(\mathcal{Q}, Q')$ is also compared to the upper bound $\overline{\Delta_f}(\mathcal{Q}, Q')$ in order to early recognize whether it is worth to keep visiting the subtree $\mathcal{T}_{Q'}$ (Line 9): the visit goes ahead only if $\Delta_f(\mathcal{Q}, Q') < \overline{\Delta_f}(\mathcal{Q}, Q')$, otherwise (i.e., if the marginal potential gain equals the upper bound) $\mathcal{T}_{Q'}$ is entirely skipped. If the subtree $\mathcal{T}_{Q'}$ is not pruned, all children Q'' of Q' are generated and, for each of them,

[3] In our implementation we avoid to consider the same reformulation multiple times by keeping trace of the *DFS code* [YH02] of each reformulation visited.

Algorithm 7 Fast_MMPG

Input: A graph database $\mathcal{D}$, a query Q, a set of reformulations $\mathcal{Q}$
Output: A reformulation $Q^* \in \mathbb{S}_Q \setminus \mathcal{Q}$ that maximizes $\Delta_f(\mathcal{Q}, Q^*)$

```
1:  Δ*_f ← −∞
2:  initialize P with {children of Q} \ Q
3:  while P is not empty ∧ max(P) > Δ*_f do
4:      Q' ← poll(P)
5:      if Δ_f(Q, Q') > Δ*_f then
6:          Q* ← Q'
7:          Δ*_f ← Δ_f(Q, Q')
8:      if |D_Q'| > 0 ∧ Δ_f(Q, Q') < Δ̄_f(Q, Q') then
9:          Q'' ← {children of Q'} \ Q
10:         add {Q'' ∈ Q'' | Δ̄_f(Q, Q'') > Δ*_f} to P
```

the corresponding upper bound $\overline{\Delta_f}(\mathcal{Q}, Q'')$ is computed according to Theorem 4.8 (Line 10). All children Q'' having $\overline{\Delta_f}(\mathcal{Q}, Q'')$ no more than the best-so-far marginal potential gain Δ_f^* are discarded, while all others are added to $\mathbf{P}$ to be processed in a later iteration (Line 11).

The children of a (reformulated) query Q' (Lines 2 and 10) are generated according to the following strategy. For each result $R \in \mathcal{D}_{Q'}$, we keep track of all the subgraphs of R that are isomorphic to Q', and we expand each of those subgraphs by one step of BFS initiated in the vertices of that subgraph. A nice side effect of this strategy is that, for each children Q'' of Q' yielded, we automatically have the corresponding result set $\mathcal{D}_{Q''}$ without running any further subgraph-search query on the database. As a result, the only subgraph-search query we need is the one to compute the results $\mathcal{D}_Q$ of the original query Q.[4]

Example 4.3. *Figure 4.4 shows the execution of the* Fast_MMPG *algorithm on an example query Q and the corresponding reformulation tree. In the example we assume that a set of reformulations $\mathcal{Q}$ have already been computed. The priority queue $\mathbf{P}$ is initialized with the children of Q, which, according to their upper bounds, follow the ordering $Q'_1 \to Q'_2 \to Q'_3 \to Q'_4$. In the first iteration, Q'_1 is extracted from $\mathbf{P}$ and the best-so-far value is set to $\Delta_f^* = \Delta_f(\mathcal{Q}, Q'_1) = 18$. Among the children of Q'_1, only Q'_{11} is added to the priority queue as the upper bound of the other child Q'_{12} is not $> \Delta_f^*$. The new ordering of the reformulations in $\mathbf{P}$ is $Q'_2 \to Q'_{11} \to Q'_3 \to Q'_4$, therefore Q'_2 is the next reformulation to be processed. The value of Δ_f^* is updated again and set to $\Delta_f^* = \Delta_f(\mathcal{Q}, Q'_2) = 20$, while all children of Q'_2 are pruned as their upper bound are less than Δ_f^*. The next reformulation processed is Q'_{11}, which leads to a new Δ_f^* value*

[4] We use the traditional subgraph-isomorphism algorithm by Ullmann [Ull76] for this. Our work is however orthogonal to the method used for answering subgraph-search queries: to achieve further speed-up, one can also resort to some existing indexing strategy [SZLY08, CKN09, CKFY11].

equal to $\Delta_f(\mathcal{Q}, Q'_{11}) = 22$. *After that, the algorithm terminates as the maximum upper-bound value within* $\mathbf{P}$ *(i.e.,* $\overline{\Delta_f}(\mathcal{Q}, Q'_3) = 21$*) becomes smaller than* Δ_f^**: the subtrees* $\mathcal{T}_{Q'_3}$ *and* $\mathcal{T}_{Q'_4}$ *are therefore entirely skipped. The reformulation ultimately outputted by* Fast_MMPG *is* Q'_{11}.

The next theorem formally shows the soundness of the proposed Fast_MMPG algorithm.

Theorem 4.9. *Algorithm 7 finds an optimal solution to the problem* $\arg\max_{Q' \in \mathbb{S}_Q \setminus \mathcal{Q}} \Delta_f(\mathcal{Q}, Q')$ *stated in Line 3 of Algorithm 6.*

Proof. Let Q^* be an optimal solution to the problem at hand derived by considering the whole search space $\mathcal{T} = \mathbb{S}_Q \setminus \mathcal{Q}$, i.e., $Q^* = \text{argmax}_{Q' \in \mathcal{T}} \Delta_f(\mathcal{Q}, Q')$. Let also $\widetilde{\mathcal{T}} \subseteq \mathcal{T}$ be the subset of the search space visited by Algorithm 7. The algorithm visits all the solutions in $\widetilde{\mathcal{T}}$, and, among these solutions, it outputs the one exhibiting maximum marginal potential gain (see Lines 5–8), i.e., it outputs a solution $\widetilde{Q}^* = \text{argmax}_{Q' \in \mathcal{T}'} \Delta_f(\mathcal{Q}, Q')$. To prove the correctness of Algorithm 7, it is thus sufficient to show that the marginal potential gain of the solution $\widetilde{Q}^*$ output by the algorithm is equal to the marginal potential gain of the optimal solution Q^*, i.e., we need to show that $\Delta_f(\mathcal{Q}, \widetilde{Q}^*) = \Delta_f(\mathcal{Q}, Q^*)$: this would clearly imply that the solution $\widetilde{Q}^*$ output by the algorithm is an optimal solution itself.

To prove this, one can easily note that the solutions that are skipped by Algorithm 7 may only come from the pruning rules in Lines 9 and 11. Thus, we need to show the correctness of these pruning rules, that is neither of these rules can leave out solutions having marginal potential gain larger than the one exhibited by the solution $\widetilde{Q}^*$ ultimately output by the algorithm.

The rule at Line 9 says that, denoting by $\mathcal{T}_{Q'}$ the subtree of the search space $\mathcal{T}$ rooted at Q', $\mathcal{T}_{Q'}$ is not visited further if the maximum marginal gain $\Delta_f(\mathcal{Q}, Q')$ of Q' is no less than the upper bound $\overline{\Delta_f}(\mathcal{Q}, Q')$ on the marginal potential gain exhibited by a solution in $\mathcal{T}_{Q'}$. Theorem 4.8 guarantees that $\forall Q'' \in \mathcal{T}_{Q'} : \Delta_f(\mathcal{Q}, Q'') \leq \overline{\Delta_f}(\mathcal{Q}, Q')$, which clearly means that *all* the solutions within $\mathcal{T}_{Q'} \setminus \{Q'\}$ cannot exhibit marginal potential gain larger than the one exhibited by Q', thus there is no need to consider them further. This proves the correctness of the pruning rule at Line 9.

The rule at Line 11 instead states that, among the children of Q', only the children Q'' such that $\overline{\Delta_f}(\mathcal{Q}, Q'')$ is larger than the best marginal potential value Δ_f^* encountered

so far are considered further. The correctness of the rule is again guaranteed by Theorem 4.8: no solutions within any subtree rooted at any of the children of Q' skipped can exhibit larger marginal potential gain than the one found so far, thus none of these children can lead to a better solution.

The correctness of the pruning rules implies that

$$\begin{aligned} &\forall Q' \in \mathcal{T} \setminus \widetilde{\mathcal{T}} : \Delta_f(\mathcal{Q}, Q') \leq \Delta_f(\mathcal{Q}, \widetilde{Q}^*) \\ \Rightarrow \quad &\Delta_f(\mathcal{Q}, \widetilde{Q}^*) = \Delta_f(\mathcal{Q}, Q^*), \end{aligned}$$

which completes the proof. □

4.6 Experimental Evaluation

Table 4.1: Characteristics of the real databases: number of graphs; *min*, *avg*, and *max* number of graph vertices/edges; number of vertex/edge labels; average density defined as $|E|/\binom{|V|}{2}$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

		# vertices			*# edges*			*# labels*		
database	*# graphs*	*min*	*avg*	*max*	*min*	*avg*	*max*	*vertices*	*edges*	*avg density*
`AIDS`	10 000	2	25	214	1	27	217	51	4	0.1
`Financial`	13 087	5	14	34	4	14	46	45	68	0.2
`Web`	17 920	2	8	48	1	7	53	639	8	0.3

In this section we empirically evaluate our approach by assessing its accuracy and efficiency on both real and synthetic databases, and comparing it to a number of baselines.

4.6.1 Experimental setting

Methods. We recall that our method corresponds to the Greedy algorithm (Algorithm 6) equipped with the proposed Fast_MMPG (Algorithm 7) to perform the marginal-potential-gain-maximization step. For the sake of brevity, in the following we denote our method by Fast_MMPG only.

We compare our Fast_MMPG to three baselines. The first baseline corresponds to the Greedy algorithm when equipped with a brute-force method to maximize the marginal potential gain, i.e., a method that visits the whole reformulation search space without exploiting any finding devised in Section 4.5.3. Such a baseline, denoted by Greedy_BF, is

mainly aimed at efficiency evaluation. The second baseline is the naïve method inspired by frequent subgraph mining and discussed in Section 4.5.1: this method selects the most frequent k supergraphs of the input query as reformulations. We refer to this baseline as k-freq. The third baseline is a method that discovers subgraph features from the graph database, indexes them, and, given a query Q, it returns the immediate (i.e., minimal) supergraphs of Q among the features present in the index as reformulations. In our implementation we extract discriminative yet frequent features according to the state-of-the-art methods defined in [SZLY08, YYH05], while we use the well-known Lindex method [YM13] for indexing such features. We refer to this baseline as Lindex.

All methods are implemented in *Java* 1.7, and the experiments are performed on a *i*686 Intel Xeon E5-2440 2.40GHz, 125GB RAM machine over Linux kernel *v*3.8.0, which we limit to 30GB in all experiments. The graph database is loaded into main memory using the ParMol library [MWU+07]. We use a copy of Lindex kindly provided by the authors of [YM13] both for the Lindex baseline and to compute the DFS codes of all methods.

Real databases. We use real-world, publicly-available databases. The main characteristics of such databases are shown in Table 4.1. Next we report a short description.

- `AIDS`[5] is a biological database extracted from the well-known *AIDS anti-viral screening* dataset[6] and traditionally used in the graph-database literature [YCHY08, SZLY08, CKN09, SLZ+10, CKFY11, LXCB12]. Vertices and edges represent atom and atom bonds, respectively.
- `Financial`[7] is a transaction workflow of loan-request processes submitted to a financial institute. Every vertex represents a subprocess while edges correspond to a resource exchanged between two processes.
- `Web`[8] is a workflow of web interactions between users and a recommender system for restaurants. A vertex is a restaurant and an edge is a browsing action.

Synthetic databases. We generate synthetic databases using the popular *GraphGen* graph generator [CKNL07].[9] We consider different database and graph sizes in order to better assess the scalability of our proposal (see Section 4.6.5).

[5] `www.cs.ucsb.edu/~xyan/software.htm`
[6] `http://dtp.nci.nih.gov/docs/aids/aids_data.html`.
[7] `www.win.tue.nl/bpi/2012/challenge`
[8] `kdd.ics.uci.edu/databases/entree/entree.html`
[9] `http://www.cse.ust.hk/graphgen/`.

Query generation. We generate random queries of various sizes. To ensure non-empty answers, we start from a vertex of a graph in the database (both sampled uniformly at random) and perform a DFS from that vertex until the desired size has been reached. To have meaningful reformulations, we discard queries having too few results (i.e., with number of results less than the number k of output reformulations). For each set of experiments and parameter configuration, we report results averaged over 10 random queries.

4.6.2 Ruling out Greedy_BF

In Figure 4.5 we report the running time of our Fast_MMPG and the Greedy_BF baseline, using a synthetic database generated with GraphGen [CKNL07], where we vary the database size and set all other parameters to their default values.

It is easy to see that Greedy_BF is not suitable for any real-world settings: on even very small databases of 30 graphs it is already four orders of magnitude slower than Fast_MMPG, and we could not run it on larger databases due to its excessive running time. For this reason, we avoid to report efficiency results for Greedy_BF in the remainder. In terms of accuracy, we recall that both Fast_MMPG and Greedy_BF employ the greedy scheme in Algorithm 6 and they both also optimally solve the sub-problem of maximizing the marginal potential gain (for the optimality of our Fast_MMPG see Theorem 4.9). As a consequence, they produce exactly the same results.

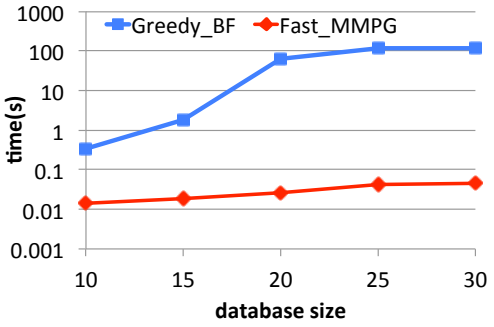
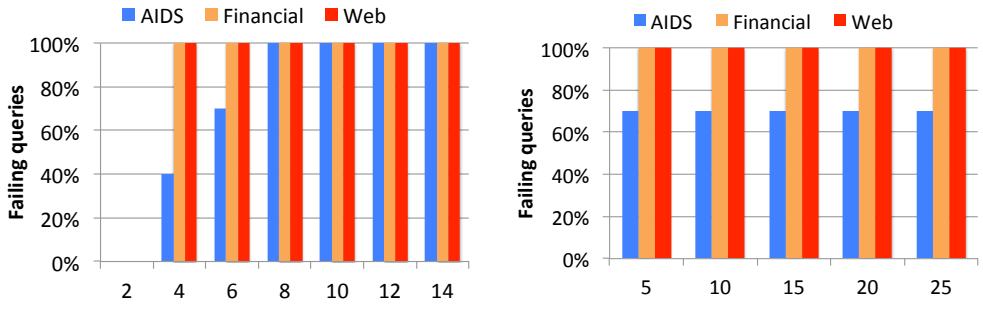


Figure 4.5: Running time of the proposed Fast_MMPG algorithm vs. the brute-force Greedy_BF baseline.

4.6.3 Ruling out Lindex

We discuss here the results achieved by the Lindex baseline. All the indexes on graph databases proposed in the literature are expressively designed to split a query graph



(a) Query size ($|Q|$) **(b)** Number of reformulations (k)

Figure 4.6: Percentage of queries for which the Lindex baseline returns no reformulations with varying (a) query size ($|Q|$), and (b) number of reformulations (k).

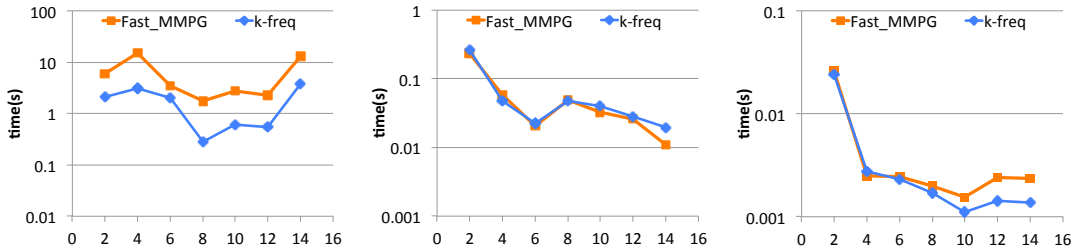
in features of smaller size, in order to speed-up subgraph-search queries. Thus, such indexes usually work well if the task is to find small-sized subgraphs of the query graph, while being less suited for the task of finding supergraphs. As a result, for queries of size exceeding the size of the largest feature in the index, the Lindex baseline would inevitably output an empty answer. As the features indexed are usually of very small size, this actually happens very often.

Indeed, Figure 4.6 shows the percentage of queries for which no reformulations are found by Lindex. It can be observed that Lindex fails in finding reformulations in most cases, e.g., 100% of the times for queries of size larger than 2 on the Financial and Web graphs. This confirms that Lindex is not really suitable for the problem of Graph Query Reformulation we tackle in this work. For this purpose, we avoid to report further details on Lindex in this sections.

4.6.4 Performance with varying parameters

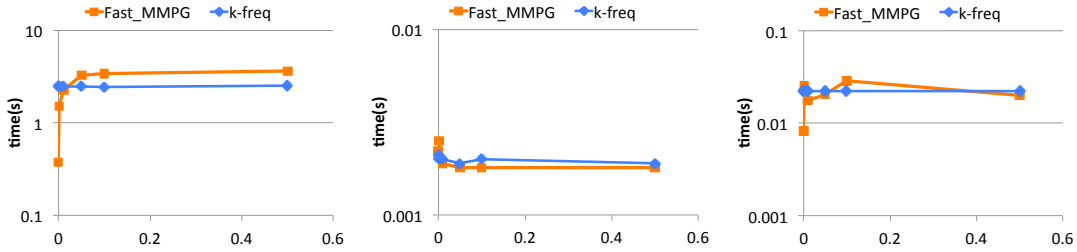
Here we discuss the results of the evaluation on real datasets. We perform tests with varying the main parameters involved in the process: (*i*) size (i.e., number of edges) $|Q|$ of the input query, (*ii*) value of the regularization factor λ (Equation (4.3)), and (*iii*) number of output reformulations k. We vary these parameters in the following ranges: $|Q| \in [2, 14]$, $\lambda \in [0, 0.5]$, $k \in [5, 25]$. While varying one parameter, we keep the other two fixed to (around) their median values, i.e., we set $|Q| = 6$, $\lambda = 0.3$, and $k = 10$.

We report running times (Figures 4.7–4.9) and quality (in terms of objective-function value, Tables 4.2–4.4) of the proposed Fast_MMPG algorithm and the baseline k-freq.



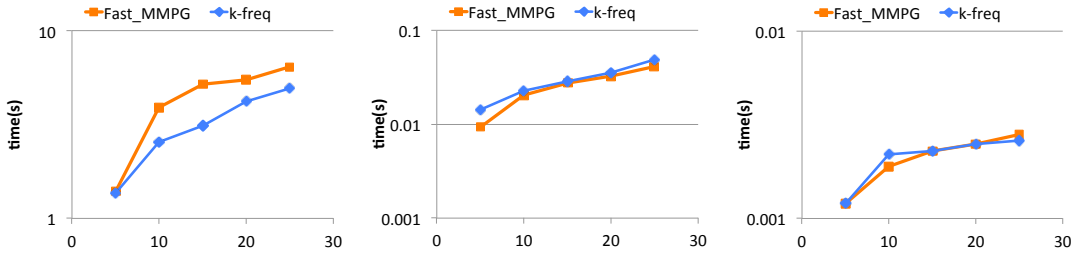
(a) AIDS (b) Financial (c) Web

Figure 4.7: Running times of the proposed Fast_MMPG algorithm and the k-freq baseline on the real datasets with varying query size $|Q|$.



(a) AIDS (b) Financial (c) Web

Figure 4.8: Running times of the proposed Fast_MMPG algorithm and the k-freq baseline on the real datasets with varying regularization factor λ.



(a) AIDS (b) Financial (c) Web

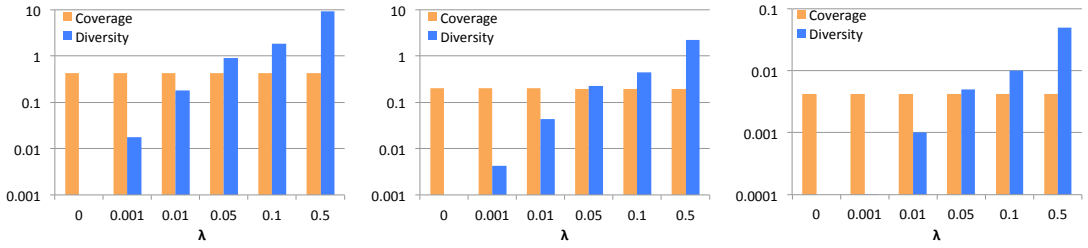
Figure 4.9: Running times of the proposed Fast_MMPG algorithm and the k-freq baseline on the real datasets with varying number of reformulations k.

Query size ($|Q|$). Figure 4.7 reports the efficiency of Fast_MMPG varying the query size. It can be observed that our algorithm can easily handle all real databases, with running times ranging from a few milliseconds (Web) to a few seconds (AIDS). The difference in time observed through the various databases is mainly due to the size of the graphs therein: the graphs in AIDS are indeed generally larger than the other two databases. Moreover, as a general finding, times are decreasing as the query size increases, which is expected as larger queries lead to smaller result sets. Finally, the running times of our method are comparable to the baseline k-freq: although this baseline employs a much simpler scheme than our Fast_MMPG, and, as such, is expected to run faster, we instead do not observe this in practice. The reason is that, even though k-freq may visit less reformulations than Fast_MMPG, the ones visited by k-freq are top-frequent

reformulations for which a quite large number of subgraph isomorphisms need to be performed. On the contrary, our Fast_MMPG algorithm does not necessarily visits top-frequent reformulations as diversity is also considered.

The quality results are shown in Table 4.2, where we report the objective-function values of Fast_MMPG and k-freq, along with the percentage gain achieved by Fast_MMPG over k-freq (last row). In general, Fast_MMPG evidently outperforms k-freq, with gain ranging from up to 40% on `Web` to 52% on `AIDS` (17% and 34% on average). The reason of the larger improvement exhibited on `AIDS` is likely due to the type of graphs contained in the database: the graphs in `AIDS` are larger and more diversified than the workflow graphs in `Financial` and `Web`, which are instead more similar to one another and thus capture less results.

Finally, we generally observe that the objective-function value decreases as the query size increases. This is expected since the objective-function value is directly proportional to the number of query results, and, clearly, the larger the query, the fewer the results.



(a) `AIDS` **(b)** `Financial` **(c)** `Web`

Figure 4.10: Values of the coverage and diversity terms (in thousands) exhibited by Fast_MMPG on real datasets with varying λ. The value of the diversity term is reported multiplied by λ.

Regularization factor (λ). The running time of Fast_MMPG with varying λ (Figure 4.8) follows a fluctuating trend for smaller values of λ, while converging for $\lambda > 0.1$. In all cases, the running time is again very small: it ranges from 3.2 milliseconds (`Web`) to 3.6 seconds (`AIDS`). Again, our Fast_MMPG is really close to the baseline k-freq.

As far as the quality results (Table 4.3), as expected, larger λ values lead to larger improvements by our Fast_MMPG over the baseline. The motivation is that a larger value of λ steers the objective function towards the maximization of diversity. Since coverage is implicitly captured, as a side effect, by the top-k frequent reformulations output by the k-freq baseline, the latter is unsurprisingly closer to Fast_MMPG for values of λ close to zero. However, we restate that a value of λ too small is not generally a

good choice, because it would practically correspond to ignore diversity, which instead plays a key role, as extensively discussed above.

Finally, we report in Figure 4.10 how the value of the coverage and diversity terms of the proposed objective function are affected by λ. As a main observation, in all databases, a good balancing between the two terms is achieved for $\lambda \geq 0.1$. A more detailed analysis on the coverage and diversity terms of our objective function (as far as the baseline k-freq too) can be found in Section 4.7.

Table 4.2: Quality (in terms of the objective function f defined in Equation (4.3), values in *thousands*) of the proposed Fast_MMPG algorithm and the k-freq baseline on real datasets with varying the query size $|Q|$.

	AIDS $\|Q\|$							Financial $\|Q\|$							Web $\|Q\|$						
	2	4	6	8	10	12	14	2	4	6	8	10	12	14	2	4	6	8	10	12	14
Fast_MMPG	30	14	6	1.5	1.8	1.6	1	34	6.4	1.6	1.5	1.1	.7	.4	.7	.1	.07	.07	.023	.021	.012
k-freq	23	11	4.3	.8	1.3	.8	.6	34	5.8	1.2	.8	1	.9	.3	.55	.09	.04	.06	.018	.019	.01
gain (%)	24	19	27	46	27	52	43	1	10	21	44	17	15	30	18	15	40	3	18	8	15

Table 4.3: Quality (in terms of the objective function f defined in Equation (4.3)) of the proposed Fast_MMPG algorithm and the k-freq baseline on real datasets with varying the objective-function regularization factor λ.

	AIDS λ					Financial λ					Web λ				
	0	0.01	0.05	0.1	0.5	0	0.01	0.05	0.1	0.5	0	0.01	0.05	0.1	0.5
Fast_MMPG	433	613	1 345	2 260	9 566	201	244	422	649	2 461	4	5.2	9.3	14.3	54.7
k-freq	409	540	1 063	1718	6 954	188	222	360	533	1 914	4	5	8.4	12.7	46.6
gain (%)	6	12	21	24	27	7	9	15	18	22	0	3	9	11	15

Table 4.4: Quality (in terms of the objective function f defined in Equation (4.3)) of the proposed Fast_MMPG algorithm and the k-freq baseline on real datasets with varying the number of output reformulations k.

	AIDS k					Financial k					Web k				
	5	10	15	20	25	5	10	15	20	25	5	10	15	20	25
Fast_MMPG	1 791	5 917	12 462	21 235	32 029	629	1 555	2 904	4 630	6 645	12	35	68	99	134
k-freq	1 373	4 336	9 709	17 061	25 667	535	1 224	2 241	3 410	5 400	7	30	62	92	123
gain (%)	23	27	22	20	20	15	21	23	26	19	41	14	8	7	8

Number of reformulations (k). The efficiency results of Fast_MMPG with varying the number of output reformulations are reported in Figure 4.9. Clearly, the running time is increasing as k increases. However, the trends on all datasets are roughly linear in k, which attests the scalability of our method with respect to the output size. Again, our Fast_MMPG is really close to the baseline k-freq.

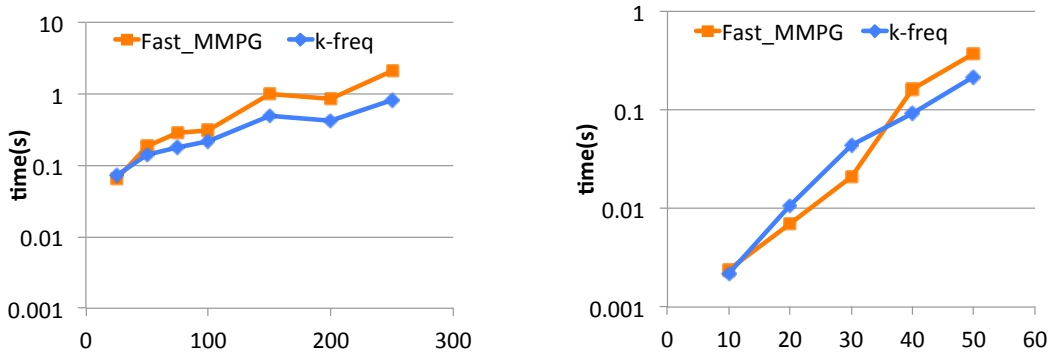
The quality results reported in Table 4.4 show that, while being better than the baseline in all settings, Fast_MMPG exhibits (slightly) decreasing gain as k increases. This is reasonable as the greedy scheme of Fast_MMPG implies that the maximum marginal-gain value gets progressively smaller as more reformulations are added to the solution.

4.6.5 Scalability

We test the scalability of our Fast_MMPG algorithm (and the k-freq baseline) on synthetic databases. Particularly, we analyze the efficiency performance from two main perspectives, that is varying (i) the database size (i.e., number of graphs), and (ii) the average size (expressed as number of edges) of the graphs in the database. To this end, we use *GraphGen* [CKNL07] to generate databases of sizes in the range [25K, 250K] and average graph sizes in [10, 50]. All other parameters are set to the default values suggested by the generator. The main parameter values are: number of vertex labels (20), number of edge labels (20), average graph density (0.3). All such values are consistent with the ones observed on our real datasets.

The results of this evaluation are reported in Figure 4.11. The main message of the left figure is that Fast_MMPG can handle query reformulation in a database of 250K graphs in a few seconds. More specifically, the trend is weakly exponential in the database size: the running time ranges from almost 0.1s (25K graphs) to 2.2s (250K graphs). This attests full scalability of Fast_MMPG on very large database sizes.

The scalability of Fast_MMPG is also confirmed by the experiment with varying the graph size: our algorithm takes less than one second for handling databases with average graph size of 50. We remark that this value is far beyond the graph size that is commonly encountered in real-world scenarios (in our three real-world databases, the average graph size is 4, 14 and 27 as reported in Table 4.1). However, here Fast_MMPG (and k-freq too) is more sensitive to changes than the previous experiment: this is expected since, like most methods on querying graph databases, our technique relies on subgraph-isomorphism-like primitives, whose running time is notoriously more affected by the size of the graph than the number of graphs, as larger graphs typically lead to more isomorphisms.



(a) database size (in thousands) **(b)** average graph size

Figure 4.11: Scalability of the proposed Fast_MMPG algorithm on synthetic databases: (a) running time vs. database size (avg graph size set to 30); (b) running time vs. average graph size (database size set to 10K).

4.6.6 Qualitative evaluation

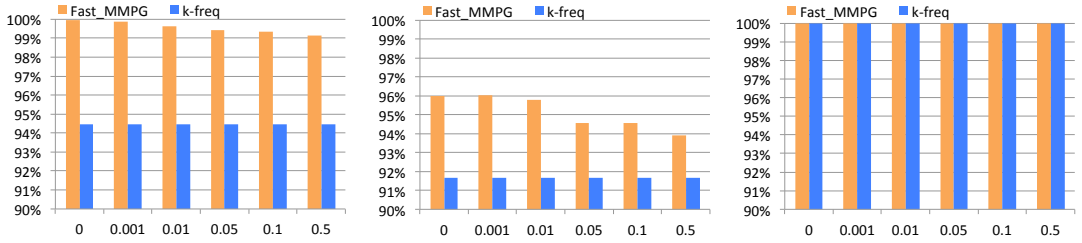
We provide here some visual examples of the results produced by our Fast_MMPG algorithm and the k-freq baseline. Figure 4.16a shows a query issued to the `AIDS` dataset. The query corresponds to a well-known chemical compound, i.e., *formaldehyde*. The reformulations output by our Fast_MMPG correspond to chemical compounds that span the search space horizontally, thus showing non-overlapping molecules, among which one can recognize two very common compounds of formaldehyde, namely *formamide* (fourth reformulation) and *acetone* (fifth reformulation). On the other hand, the k-freq reformulations are very general and all close to each other (some of them are even subgraphs of other reformulations, e.g., first and fourth reformulation): such reformulations are therefore much less informative than the ones found by our Fast_MMPG.

The second example on the `Financial` dataset (Figure 4.16b) clearly shows that methods based only on frequency (like k-freq) are not suitable for capturing the various (diverse) alternatives from the results of a query. Indeed, once a frequent structure (reformulation) containing the query has been encountered, all other reformulations are most likely generated starting from there, meaning that every subsequent reformulation is a supergraph of the previous one. This is exactly what happens with the reformulations output by k-freq for the example in Figure 4.16b. Instead, our Fast_MMPG returns reformulations whose common structure mostly corresponds to the query itself.

4.7 Impact of coverage and diversity on the Graph Query Reformulation problem

In this section we provide further insights on the two main terms of the proposed objective function f, i.e., coverage (*cov*) and diversity (*div*). In Figure 4.12 and Figure 4.13 we report the values of these two terms exhibited by the proposed Fast_MMPG and the k-freq baseline, on the various datasets.

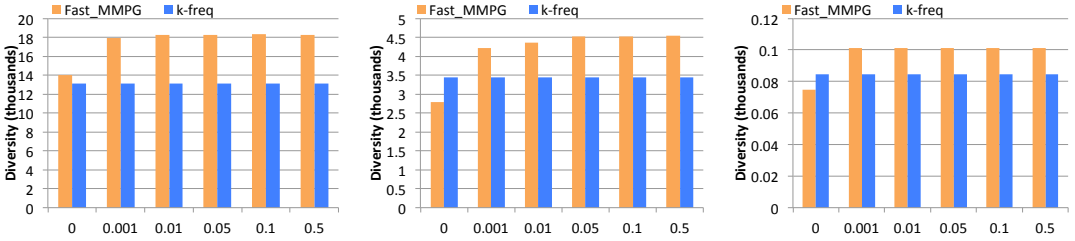
Figure 4.13 shows that the diversity exhibited by our Fast_MMPG is evidently higher than that of k-freq, for all values of λ but $\lambda = 0$, which corresponds to considering only coverage in the proposed objective function f (note the values reported are expressed in thousands). This is clearly expected as the baseline k-freq does not explicitly account for diversity.



(a) `AIDS` **(b)** `Financial` **(c)** `Web`

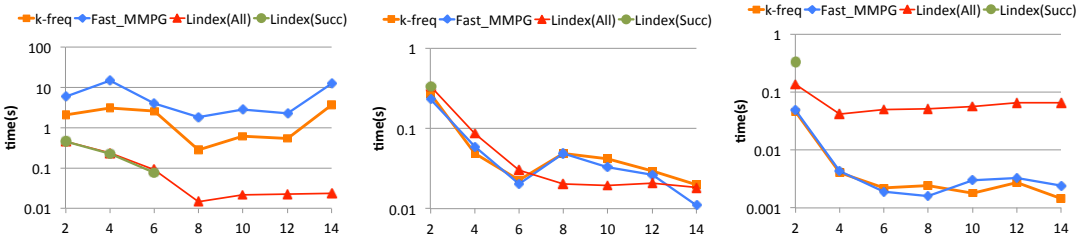
Figure 4.12: Value of the coverage term (in percentage) exhibited by the proposed Fast_MMPG and the k-freq baseline on real datasets with varying λ.

A more interesting finding is what arises from Figure 4.12, where it is shown that our Fast_MMPG is generally better than the baseline in terms of coverage too, even for larger values of λ. The reason of this behavior is the following. Although coverage is somehow implicitly taken into account by frequency (which is the (only) aspect considered by k-freq), the latter leads, as a side effect, to favor reformulations that come all from the same "heaviest" branch of the reformulation tree, i.e., the branch that contains the most frequent reformulation. In other words, k-freq tends to recursively expand the branch containing such most frequent reformulation, so that all the reformulation outputted are supergraphs of such most frequent reformulation, and, as such, are not able to cover other more diverse results. This is also the reason why k-freq generally finds reformulations that are very similar to each other, and thus not really interesting. A practical example of this behavior (on the `Financial` dataset) has already been reported and discussed in Section 4.6.6, Figure 4.16.



(a) AIDS (b) Financial (c) Web

Figure 4.13: Value of the diversity term exhibited by the proposed Fast_MMPG and the k-freq baseline on real datasets with varying λ. Values are expressed in thousands.



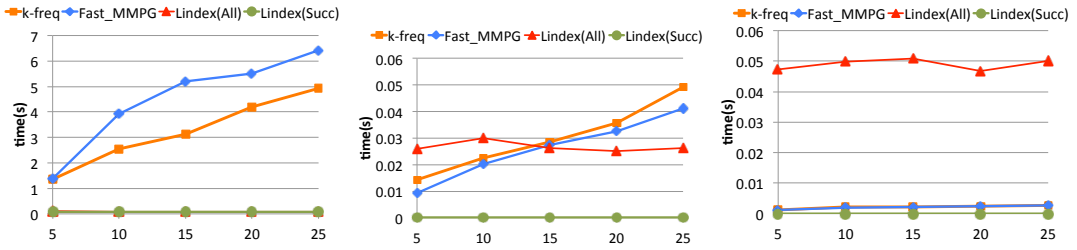
(a) AIDS (b) Financial (c) Web

Figure 4.14: Running times of the proposed Fast_MMPG algorithm and the baselines on the real datasets with varying query size $|Q|$.

4.8 Comparison with Index Based Methods [Extended]

We compare here time and quality performance of the proposed Fast_MMPG algorithm to the Lindex baseline. Figures 4.14–4.15 show running times on the three datasets with varying query size and number of reformulations, respectively. For Lindex, we report both the time when the baseline returns a non-empty set of reformulations (Lindex(Succ)) and the overall time even though no reformulation is output (Lindex(All)). As expected Lindex is in general faster that Fast_MMPG as it can exploit an indexing structure and the fact that most queries are actually not executed as no results are returned.

Nevertheless, in terms of quality, Tables 4.5–4.7 clearly show that Lindex performance is rather poor when compared to our Fast_MMPG, even in those few cases where Lindex is able to produce non-empty answers.



(a) AIDS **(b)** Financial **(c)** Web

Figure 4.15: Running times of the proposed Fast_MMPG algorithm and the baselines on the real datasets with varying number of reformulations k.

Table 4.5: Quality (in terms of the objective function f defined in Equation (4.3), values in *thousands*) of the proposed Fast_MMPG algorithm and the Lindex baseline on real datasets with varying the query size $|Q|$. The first two lines refer to absolute values, while third and fouth lines refer to values averaged by k.

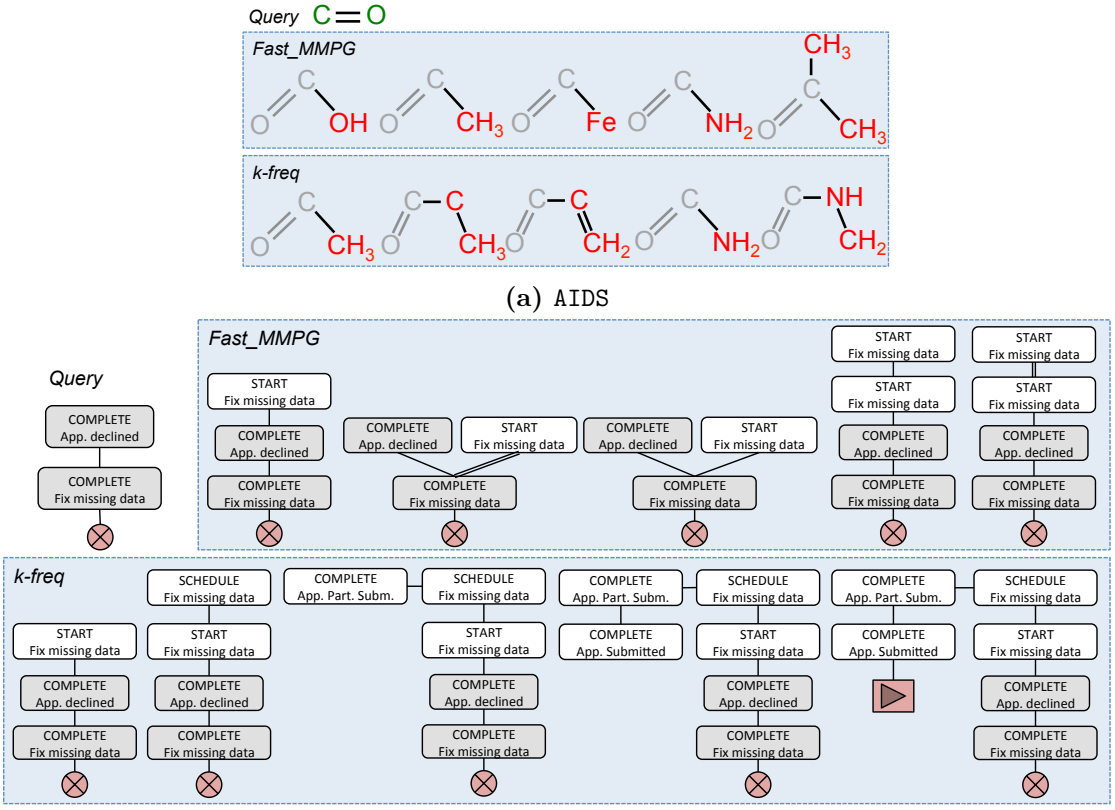
	AIDS $\|Q\|$							Financial $\|Q\|$							Web $\|Q\|$						
	2	4	6	8	10	12	14	2	4	6	8	10	12	14	2	4	6	8	10	12	14
Fast_MMPG	30	14	5.9	1.5	1.8	1.6	1	34	6.4	1.6	1.5	1.1	.7	.4	1.3	.17	.04	.07	.05	.03	.01
Lindex	.03	.01	0	0	0	0	0	.02	0	0	0	0	0	0	1	0	0	0	0	0	0
Fast_MMPG	3	1.4	.6	.15	.18	.16	.11	3.4	.64	.16	.15	.12	.07	.04	.13	.02	.01	.01	.01	0	0
Lindex	2.5	.7	.15	0	0	0	0	1.7	0	0	0	0	0	0	.1	0	0	0	0	0	0
empty answers (%)	0	40	70	100	100	100	100	0	100	100	100	100	100	100	0	100	100	100	100	100	100

Table 4.6: Quality (in terms of the objective function f defined in Equation (4.3)) of the proposed Fast_MMPG algorithm and the Lindex baseline on real datasets with varying the objective-function regularization factor λ. The first two lines refer to absolute values, while third and fouth lines refer to values averaged by k.

	AIDS λ					Financial λ					Web λ				
	0	0.01	0.05	0.1	0.5	0	0.01	0.05	0.1	0.5	0	0.01	0.05	0.1	0.5
Fast_MMPG	430	613	1 344	2 259	9 574	430	613	1 344	2 259	9 574	430	613	1 344	2 259	9574
Lindex	229	271	442	654	2 356	229	271	442	654	2 356	229	271	442	654	2356
Fast_MMPG	43	61	134	226	957	43	61	134	226	957	43	61	134	226	957
Lindex	23	27	44	65	236	23	27	44	65	236	23	27	44	65	236
empty answers (%)	70	70	70	70	70	100	100	100	100	100	100	100	100	100	100

Table 4.7: Quality (in terms of the objective function f defined in Equation (4.3)) of the proposed Fast_MMPG algorithm and the Lindex baseline on real datasets with varying the number of output reformulations k. The first two lines refer to absolute values, while third and fouth lines refer to values averaged by k.

	AIDS k					Financial k					Web k				
	5	10	15	20	25	5	10	15	20	25	5	10	15	20	25
Fast_MMPG	1 791	5 917	12 462	21 235	32 029	629	1 555	2 904	4 630	6 645	12	35	68	99	134
Lindex	682	1 505	1 505	1 505	1 505	0	0	0	0	0	0	0	0	0	0
Fast_MMPG	358	592	831	1 062	1 281	126	156	194	231	266	2	3	5	5	5
Lindex	136	151	100	75	60	0	0	0	0	0	0	0	0	0	0
empty answers (%)	70	70	70	70	70	100	100	100	100	100	100	100	100	100	100



(a) `AIDS`

(b) `Financial`

Figure 4.16: Examples queries on the `AIDS` and `Financial` datasets and the reformulations returned by Fast_MMPG and k-freq ($k = 5$).

4.8.1 Summary of experimental assessment

Our experiments provide evidence that Fast_MMPG is a better choice than the two baselines to solve Problem 1, both in terms of accuracy and efficiency. On the one hand, while Greedy_BF is guaranteed to provide the same quality of Fast_MMPG, its poor efficiency makes it not suitable for any real-world graph database. On the other hand, the method based on frequent subgraph mining, i.e., k-freq, while being comparable to Fast_MMPG in terms of efficiency, it is evidently less accurate.

The quality of the reformulations generated is guaranteed by Theorem 4.3. A visual inspection of the results confirms the practical usefulness of query reformulation in graph databases. For instance, Figure 4.16 represents real query reformulations generated by our method in the `AIDS` and `Financial` datasets. In terms of scalability, the evaluation on datasets with up to 250K graphs and graph size up to 50 reports time < 1 sec: this makes our method Fast_MMPG a good candidate for real-world deployment.

Chapter 5

Exemplar Query Answering

So far, we have described two methods to scope with underspecified/overspecified queries. In this chapter we deal with the case of users that are not aware of the correct conditions to formulate the query. We introduce the novel *exemplar queries* paradigm, in which the user query is considered an example of the intended results. Although exemplar queries find applications in various data-models, we propose a solution for labeled graphs and, in particular, knowledge graphs. The proposed solution scales on incredibly large graphs, returning real time answers. We experimentally evaluate our approach in terms of quality and time. We also propose a user study to support the adoption and usefulness of exemplar queries.

5.1 Contributions

Our contributions can be summarized as follows.

- We introduce and formally define a novel form of query answering, referred to as *exemplar queries*, that treats a query as a sample from the desired result set.
- We study exemplar queries for graph-based models, and devise two similarity measures, based on subgraph isomorphism and strong simulation, that take edge-labels into consideration. In addition, we provide a theoretical analysis for our measures, proving their correctness, and we demonstrate that the two proposed similarity measures capture different, yet interesting use cases.

- We propose two algorithms to compute the exact solution: a straightforward solution, and an optimized algorithm that can prune the search space. Furthermore, we describe an approximation algorithm with significant efficiency gains and minimal effect on quality, which can be used for real-time query answering.

- We perform a thorough experimental evaluation, using the largest multigraph ever used (Freebase) in this field. We experimentally show that existing approaches either fail to produce correct exemplar query evaluations, or they do so in a much longer time, that makes them inapplicable for online applications. In contrast, the experiments demonstrate the efficiency of our solutions, and a user-study validates the usefulness of exemplar queries.

5.2 Outline

This chapter is organized as follows. We first introduce in Section 5.3 the motivation of exemplar queries and highlight the practical applications in the every day life. Section 5.4 introduces exemplar queries and the exemplar query answering problem, proposing a natural application on knowledge graphs. In Section 5.5 we propose two instances of the similarity function used to answer an exemplar query, namely subgraph isomorphism and strong simulation, as well as efficient algorithmic solutions that scale to large graphs. Section 5.6 describes a natural ranking function that exploits both structural and distance-based measures. Finally, in Section 5.7 we present an experimental assessment of our algorithms in terms of quality and performance.

5.3 Motivating example

Consider a student who wants to perform a study on company acquisitions in the Bay area, without being an expert in the field, or familiar with the related terminology. Writing a query with the terms “acquisitions” and “Bay Area” will return documents talking about acquisitions and also mentioning the Bay area. An article on the takeover of del.icio.us by Yahoo! may not be returned if the actual words of acquisition and Bay area are not explicitly mentioned in the text.

The student knows that a good case of the type of acquisition she is looking for is the one of YouTube by Google. Thus, she issues the query: *"Google founded-in Menlo Park acquired YouTube"*. The search engine typically responds with results related to Google, Menlo Park, and YouTube, but will not return anything related to an acquisition of del.icio.us by Yahoo!. If many users have performed similar searches in the past, an analysis of the query logs may reveal that information and the search engine (based on log analysis) may propose, in the related searches section, queries on Yahoo! and del.icio.us. (A simple test in existing search engines reveals that this is not actually happening.) Relaxing one or more of the query conditions does not help in a significant way, since the results are still focused around the Google case.

Consider now a second candidate answer for the user query: Paramount that was acquired by CBS. Among the Yahoo!-Tumblr and CBS-Paramount, it is more likely that the former is among the company acquisitions that the user is interested in, and not the latter. This is because even though Yahoo! was founded in a different city than Google, that city is still in California (just like with Google), while the city that CBS was founded is in New York. Furthermore, the example of Google-YouTube that the user provided is about IT companies, and so are the Yahoo!-Tumblr, while CBS-Paramount belong to the brodcasting industry.

Thus, there is a need to devise a method for inferring the set of elements that the user is interested in from a sample (of that set), provided by the user.

5.4 Background and Problem

Achieving the required functionality can be seen as a two-step process. The first is to identify in the data repository the structures satisfying the specifications in the user query, i.e., those that represent the sample that the user already knows to be part of the desired result set. This can be easily achieved using traditional query evaluation techniques. We denote the results of this type of evaluation of a query Q as $\mathcal{D}_Q$ and refer to it as the *user sample*.

The second step is to find the remaining structures of interest for the user based on the structure that has been identified in the first step. Note that there exists a query that describes all these structures that the user is looking for, it is just that she is not aware

of that query, or is not in a position to describe it. Thus, it is natural to assume that all the structures of interest have some commonalities, especially to the one that the user provided as an indicative example. As such, we are interested in finding similar structures to the results of the first step, and return these results as an answer to the user-provided query.

We refer to this new query paradigm as *exemplar queries* and the results of their evaluation as *relevant answers*.

Definition 5.1. The evaluation of an *exemplar query* Q_e on a database $\mathcal{D}$, denoted as $xmpEval(Q_e)$, is the set $\{a \mid \exists s{\in}\mathcal{D}_{Q_e} \wedge a{\approx}s\}$, where a and s are structures in $\mathcal{D}$ and the symbol $\approx$ indicates a similarity function.

Looking for structures similar and not exact to those explicitly described in the query, reminisces query relaxation. Yet, our problem is different. In query relaxation, one or more of the query conditions are relaxed, so the results in the answer set are elements that satisfy fewer conditions of the user query. The desired results in our case may be based on characteristics that are different to those mentioned in the user query, therefore query relaxation fails to answer exemplar queries.

Note that the definition of exemplar queries is independent of the data model, the query form, the retrieved results and the similarity function. As long as there is a standard query evaluation methodology and some similarity function that can be used to fit a specific use case, the exemplar queries can be answered. This leads to flexibility and the ability to use exemplar queries in a wide range of different applications. We are particularly interested in applying exemplar queries in cases where the data is highly heterogeneous and have some relaxed structure. For that reason we have chosen to use a flexible data model, a simple query form with a traditional query evaluation that is based on subgraph matching, and a very generic similarity function that is based on edge label-preserving similarity on graphs.

We assume an infinite set of labels $\mathcal{L}$ and of values $\mathcal{V}$. The set $\mathcal{V}$ consists of an infinite set of atomic values $\mathcal{T}$ and of object identifiers $\mathcal{O}$, i.e., $\mathcal{V}{=}\mathcal{T}{\cup}\mathcal{O}$. An *object* is a representation of a real world entity or concept and is modeled through an object identifier and a set of attributes for that identifier that model characteristic properties of the real world entity or concept.

Definition 5.2. A *database* $\mathcal{D}$ is a graph $\langle N, E\rangle$ where $N \subseteq \mathcal{O}$ and $E \subset \mathcal{O}\times\mathcal{L}\times(\mathcal{O}\cup\mathcal{T})$, both finite.

The expression $n \xrightarrow{\ell} n'$, denotes an edge from node n to node n' labeled ℓ. We also say that two nodes n_1, n_2 are *equivalent*, and denote it as $n \equiv n'$, if they represent the same atomic value or the same object, i.e., the identifiers of the objects they respectively represent are the same.

A query is traditionally an expression describing a set of objects alongside a set of conditions they need to satisfy. These conditions describe certain characteristics of these objects and the relationships they may have among them. We make the natural assumption that the objects referenced in a query are somehow all connected, otherwise the query expression would actually constitute two independent queries. Answering a query means finding the subgraphs in the database that have a structure matching the graph representation of the query. The set of these subgraphs constitutes the answer set of the query.

Definition 5.3. A *query* Q is a database whose graph representation is a connected graph $Q : \langle N_Q, E_Q\rangle$. An *answer* to a query $Q : \langle N_Q, E_Q\rangle$ on a database $\mathcal{D}$ is any connected subgraph $\mathcal{D}' : \langle N_{\mathcal{D}'}, E_{\mathcal{D}'}\rangle$ of $\mathcal{D}$ matching the query Q, i.e., there exists a non-trivial binary relation $\mathcal{R}$, such that $\forall n_Q \in N_Q, \exists n_{\mathcal{D}'} \in E_{\mathcal{D}'} : n_Q \mathcal{R}\ n_{\mathcal{D}'}$. The set of all such subgraphs, denoted as $\mathcal{D}_Q$, is referred to as the *answer set* of the query.

Note that this type of query evaluation can also accommodate cases where user provided queries are flat keyword queries, provided that they are first translated to some structured form in accordance with Definition 5.2. This task is outside the scope of this work, but there is already a large amount of literature [BDG+11] on that topic.

Any choice of the relation $\mathcal{R}$ is admissible, however we assume to have one that preserves the semantics of the user query.

Query Q is finally evaluated as an exemplar query through Definition 5.1. As such, in this chapter we tackle the following problem.

Problem 2 (Exemplar Query Answering)**.** *Given an exemplar query Q_e, find all answers $a \in \mathcal{D}$ such that $a \in xmpEval(Q_e)$ for a chosen similarity function $\approx$.*

Regarding the similarity function, although multiple different forms of similarity can be used, we consider two alternatives based on edge-preserving subgraph matching relations. The first is a similarity function based on the notion of *subgraph isomorphism* [Coo71]. The second is a more elastic function, based on the recently introduced *strong simulation* [MCF+14]. While subgraph isomorphism searches for perfect matches of the input query in the database, strong simulation groups matching nodes based on the edge-labels of the neighbor nodes. As we discuss in more detail later on, strong simulation relaxes the strict requirements of isomorphism, while preserving the topology and the semantics of the original query. Finally, we note that we are interested in returning a ranked list of results, and in particular the top-k most similar and relevant structures.

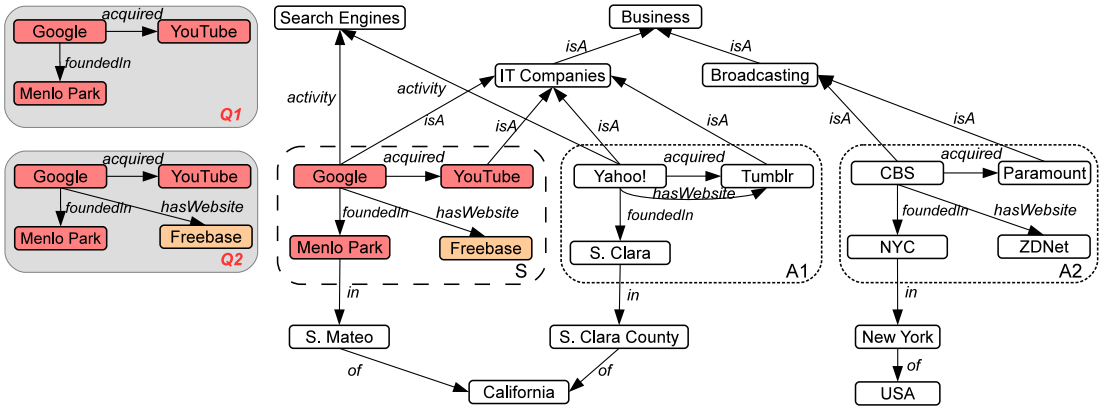


Figure 5.1: An instance of the Exemplar Query Answering problem.

Example 5.1. *Consider the example described in Section 5.3 and the portion of the database illustrated in Figure 5.1. The user query (the exemplar query) is Q1 shown at the top left corner of the figure. The evaluation of that query on the database results to the user sample that is indicated in the database with the dashed box labeled S. Searching for similar structures (edge-isomorphic structures) to this user sample, results to the two structures indicated with the dotted line boxes labeled A1 and A2, that serve as the relevant answers to the exemplar query. Among the two relevant answers, the neighborhood of A1 has more nodes and edges in common to the user sample S, for instance, the* `IT Company`*, the* `Search Engine` *and the* `California`*, than those that the neighborhood of A2 has in common, hence, A1 should be ranked higher than A2.*

Now consider query Q2 where the user, differently from Q1, also asks for companies that owns a website. The query matches the sample S but the only perfect match is A2, that is instead semantically further away than A1. In this case, the user does not necessarily need strict equality, but is interested in companies with at least *one acquisition and*

one website. Using simulation both $A1$ and $A2$ would be returned as answers, since `Tumblr`*, being both an acquisition and a website, satisfies the requirements of the query. Therefore, in this example* `Tumblr` *serves the purpose of both* `Freebase` *and* `YouTube`*. This motivates the study of both similarity functions.*

Since the first step of the exemplar query evaluation is a standard search in a graph for a subgraph matching the user query and many solutions have already been studied [PHIW12, KA11, KRS+09], we will not discuss this problem further. Instead, we focus on the implementation of the second step, which is to devise a method that given such subgraph (the user sample) finds other edge-isomorphic (or alternatively simulating) subgraphs (the relevant answers) and ranks them based on their neighborhood. One of the main challenging parts of this is that there is no clear limit on how large a neighborhood to consider, apart from the entire database. In our implementation, we use Freebase, which is one of the largest knowledge-graphs available nowadays. Existing works on graph similarity concentrate the effort of searching on a large number of small graphs, but searching on a very large graph in the form and size we consider here has not been considered before, even though there is an increasing interest for such application [LMP+14].

5.5 Algorithmic Solution

5.5.1 The Basic XQ Algorithm

Once the user query has been evaluated and the *sample* S has been identified in the database $\mathcal{D}$, the set of similar structures will have to be discovered. To do so, the user sample S will have to be compared with every other subgraph in the database. Instead of considering the exponential number of subgraphs in the database, a node n_s from S is randomly selected to serve as a seed. Then all the nodes in the database $\mathcal{D}$ are considered, one at a time. For each such node n, we check whether a subgraph that contains n and is similar to S can be constructed when mapping n_s to n. If such a graph is found, then it is added in the result set, i.e., the set of *relevant answers*. At the end of this procedure the relevant answers are sorted and returned to the user (all of them, or only the top-k) as the result to the exemplar query. (The sorting task is studied in detail in Section 5.6.)

Algorithm 8 XQ

Input: Database $\mathcal{D} : \langle N, E \rangle$
Input: User Query Q
Output: Set of relevant answers $\mathfrak{Q}$
1: $\mathfrak{Q} \leftarrow \emptyset$
2: $S \leftarrow eval(Q)$
3: $n_s \leftarrow selectARandomNode(S)$
4: **for each** $n \in N$ **do**
5: $\quad A \leftarrow$ FindSimilarSubgraphs$(S, n_s, \mathcal{D}, n)$
6: $\quad$ **if** $A \neq \emptyset$ **then**
7: $\quad\quad \mathfrak{Q} \leftarrow \mathfrak{Q} \cup A$
8: $Rank(\mathfrak{Q})$
9: **return** $\mathfrak{Q}$

The pseudo-code of the above steps is described in Algorithm 8. The construction of the matching subgraphs (line 5 in Algorithm 8) is done by initially considering a graph G consisting only from the node n_s and a subgraph T consisting only from node n, and assuming that a similarity relation maps n_s to n. Then the algorithm iteratively tries to expand the subgraphs G and T with edges from S and $\mathcal{D}$, respectively, such that the resulting subgraphs remain similar (based on the selected similarity function). If (after a number of steps) the graph G becomes equal to S, then T is one of the answers.

Searching for possible matches of the user sample in the entire database, as the Algorithm XQ requires, is an expensive operation. Thus, one of the main challenges is how to effectively and efficiently reduce the search space preserving quality guarantees on the answers. In what follows we propose solutions based on structural properties of the database that adapt to different similarity functions.

5.5.2 Instantiations of the similarity function

The XQ algorithm requires the definition of a similarity function to find the answers to an exemplar query. Although different similarity function could fit the definition, we are interested in those that preserve the structural and semantic properties of the user query. More specifically, a candidate similarity function should preserve the edge-labels of the user sample, and the (basic) connections between nodes. We identify two compelling similarity functions, based on: (1) *subgraph isomorphism*, which seeks for exact matches and is known to be **NP**-hard, and (2) *strong similarity*, a weaker notion of subgraph matching that admits a cubic-time solution in the size of the query.

In the following sections, we formally define the two similarity functions we use in this work, and discuss their properties.

Subgraph isomorphism. The most natural definition of similarity to the query terms is strict equality. In graph terms, this means finding structures that are subgraph isomorphic to the user sample. While subgraph isomorphism is defined over node and edge labels, matching node labels means referring to the exact same object, which is too strict for the exemplar query scenario. Therefore, we define edge-preserving[1] *isomorphism* as follows:

Definition 5.4. A database $\mathcal{D}$ is edge-preserving *isomorphic* to a database $\mathcal{D}'$, denoted as $\mathcal{D} \simeq \mathcal{D}'$, if there is a bijective function μ from the nodes of $\mathcal{D}$ to the nodes of $\mathcal{D}'$ such that for every edge $n_1 \xrightarrow{\ell} n_2$ in $\mathcal{D}$, the edge $\mu(n_1) \xrightarrow{\ell} \mu(n_2)$ is in $\mathcal{D}'$.

Isomorphism is a very restrictive similarity function, in that it recognizes only exact structures. We acknowledge that this level of precision could be desired in some cases but detrimental in other settings. Consider for instance the query $Q2$ and the two graphs S and $A2$ from Example 5.1. They are conceptually very close, S is an IT company that has bought another company, owns a website and was founded in California. $A1$ on the other hand differs from S, because Tumblr is a website and also an acquisition of Yahoo. However, the user could be interested in $A1$ and may want it included in the results. This more flexible similarity requires the notion of *simulation* [Par81], and in particular, *strong simulation* [MCF+14]. ■

Strong simulation. Intuitively, a graph simulates a query graph if it mimics the same edge sequences of the query and, as such, it preserves sequences of edge labels in the same order. In practice, this translates in checking if any sequence of edge labels in the query is contained in the graph. Since the subgraph match is performed in a sequence-wise fashion, this notion preserves the same semantics of the query, yet allowing for some freedom in the structure.

Definition 5.5 (Simulation)**.** Let $S : \langle N_s, E_s \rangle$ and $\mathcal{D} : \langle N, E \rangle$ be two databases represented as graphs. $\mathcal{D}$ *simulates* S if there exists a relation $\mathcal{R}$, such that, for every node $n_s \in N_s$ and $n \in N$ for which $(n_s, n) \in \mathcal{R}$ and $n_s \xrightarrow{\ell} n'_s$, there exists a n' such that $n \xrightarrow{\ell} n'$ and $(n'_s, n') \in \mathcal{R}$.

[1] In the rest of the document we will be dropping the part "edge-preserving".

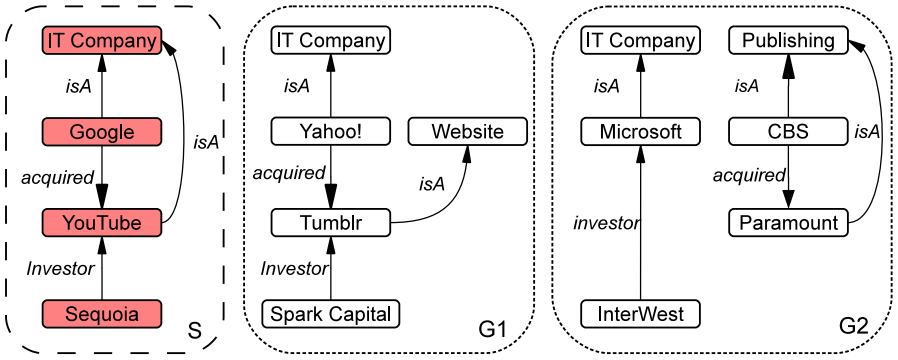


Figure 5.2: A sample (S) and two simulating graphs ($G1$ and $G2$).

This flexibility though, has some significant shortcomings. First, graph simulation does not consider parent-child relationships, since it only requires that nodes in the relation match the outgoing edges of the query. Second, the matched graph is not bounded, in that any sequence of edges with the same label can be matched to a single edge in the query. These issues are illustrated in the following example.

Example 5.2. *Consider the scenario in Figure 5.2. The user is asking for IT companies for which we know the venture fund, such that one acquired the other. She provides the example "Sequoia Capital invested in IT company Youtube acquired by IT company Google". This is depicted as the sample S in the figure, and we then want to find similar graphs using simulation. Searching for similar structures results in $G1$ and $G2$. $G1$ simulates S because both Google and Yahoo! have the same* acquired *edge, Youtube and Tumblr have an* isA *edge, then IT Company matches both IT Company and Website in $G1$, and Sequoia and Spark matches for the* investor *edge. However, the fundamental property that the two companies are of the same type is lost, since simulation does not require to match nodes with the same parent. Note that also $G2$ simulates S, since Google is matched by Microsoft and CBS, YouTube is matched by Paramount, IT Company is matched by Publishing and IT Company in $G2$, and Sequoia is matched by InterWest. Intuitively, any edge sequence in S is matched by some sequence in $G2$. Simulation disregards the locality of the match, finding possible answers in any place of the graph.*

Motivated by the above discussion, we adopt a more stringent similarity function called *strong simulation*. Strong simulation requires the definition of *dual simulation*. Dual simulation is a bidirectional simulation that checks both the incoming and outgoing edges of each query node.

Definition 5.6 (Dual simulation)**.** Let $S : \langle N_s, E_s \rangle$ and $\mathcal{D} : \langle N, E \rangle$ be two databases represented as graphs. $\mathcal{D}$ *dual simulates* S, denoted as $S \trianglelefteq_D \mathcal{D}$, if there exists a relation $\mathcal{R}$, such that for every node $n_s \in N_s$ and $n \in N$ for which $(n_s, n) \in \mathcal{R}$:

(1) for all $n_s \xrightarrow{\ell} n'_s$, exists n' such that $n \xrightarrow{\ell} n'$ and $(n'_s, n') \in \mathcal{R}$, and

(2) for all $n''_s \xrightarrow{\ell} n_s$, exists n'' such that $n'' \xrightarrow{\ell} n$ and $(n''_s, n'') \in \mathcal{R}$.

While dual simulation admits answers of any diameter, strong simulation is bounded to the diameter of the query. Strong simulation is based on the notion of neighborhood. We call d-neighbor of a node n a node that is reachable from n in at most d steps, i.e., the shortest path from n to this node is no longer than d.

Definition 5.7 (d-neighbor)**.** Let $n \in N$ be a node of a database $\mathcal{D} = \langle N, E \rangle$. The node $n_i \in N$ is a d-neighbor of n if there exists a shortest path from n to n_i of length at most d. The d-neighborhood of n, denoted as $\mathbb{N}_d(n)$, is the set of d-neighbors of n. The d-graph of n, denoted as $\mathcal{D}[n, d]$ is the subgraph of $\mathcal{D}$ induced[2] by the nodes in $\mathbb{N}_d(n)$.

Strong simulation defines bounds on the size of the simulation. Moreover, as proved in [MCF+14], the size of the maximum dual simulation relation is bounded by the diameter of the query. Recall that the diameter of a query is the length of the longest shortest path.

Definition 5.8 (Strong simulation)**.** A database $\mathcal{D} : \langle N, E \rangle$ *strong simulates* a database $S : \langle N_s, E_s \rangle$, denoted as $S \trianglelefteq_S \mathcal{D}$, if there exists a node $n \in N$ and a d-graph $\mathcal{D}[n, d]$ such that:

(1) d is equal to the diameter of the query S.

(2) $S \trianglelefteq_D \mathcal{D}[n, d]$ with the maximal dual simulation (i.e., any other dual simulation of S in $\mathcal{D}[n, d]$ is contained in the maximal).

This definition embodies the two important properties of bounding the simulation relation size within the d-graph, and preserving the parent-child relationships. Looking back at Example 5.1, we observe that using Definition 5.8, the query returns both $A1$

[2] A subgraph induced by a set of nodes N is the subgraph whose edges have both endpoints in N.

and $A2$ as results. At the same time, both $G1$ and $G2$ (shown in Figure 5.2) are rejected, which is the desired behavior.

Note that, differently from [MCF+14], our definition matches edge labels instead of node labels. In Section 5.5.4, we describe how we can adapt the algorithms in [MCF+14] for this case, providing analytical results on the correctness of this adaptation. ■

The following sections introduce algorithmic solutions for both similarity functions. Section 5.5.3 introduces approximate and exact algorithms to evaluate exemplar queries with isomorphism, while Section 5.5.4 describes our strong simulation algorithms, designed for the case of exemplar queries.

5.5.3 Finding Subgraph Isomorphic Answers

Since subgraph isomorphism is an **NP**-hard problem, we need to carefully design our algorithmic solutions in order to be efficient in practice. This becomes particularly important when the database size is large. In this section, we present the methods, and the ideas behind them, that we devised in order to efficiently answer exemplar queries using isomorphism as the similarity measure.

5.5.3.1 An Efficient Exact Solution

To improve the performance, we propose an effective way to prune the search space, i.e., the list of database nodes we have to match to the nodes of the user sample in order to find isomorphic structures, leading to a new algorithm: FastXQ. The FastXQ algorithm is divided into two steps, first we use the query to drive a process that will restrict and prune the search space, then we apply XQ to the resulting restricted spaced. To prune the space we devise an efficient technique for comparing nodes, and an algorithm for effectively rejecting pairs of nodes that are bound to not participate in any isomorphic mapping, we call this algorithm IterativePruning. Although this technique may lead to false positives, the schema is effective and reduces significantly the search space. The false positives are subsequently removed by running the traditional isomorphic verification algorithm on them.

To compare nodes, inspired by [KLY+11], we devise a technique that is meant to represent the neighborhood in a compact way, and to match the nodes in advance without the need to examine all the nodes in the graph. In more details, the idea is to store in advance a compact representation of the neighborhood of each node, i.e., nodes and edges that are at a fixed distance d from each node. This provides an effective way to compare nodes, allowing the pruning to remove the non-matching nodes without having to actually visit their neighborhood.

A basic concept of our approach is also the notion of neighborhood introduced in Definition 5.7.

For every node in the database we compute a table consisting of the number of nodes that are reachable from that node at some specific distance and with a path ending with a label ℓ. In other words, for a node n, for every label ℓ and for every distance i we keep the cardinality of the set $W_{n,\ell,i}$, where

$$W_{n,\ell,i} = \{n_1 | n_1 \xrightarrow{\ell} n_2 \vee n_1 \xleftarrow{\ell} n_2, n_2 \in \mathbb{N}_{i-1}(n)\}$$

In practice, since doing so for every node in the database is expensive in terms of space, we employ an implementation similar to the idea of the inverted indexes. We use an index structure that for every label and for every distance can provide a list of all the nodes that have a label ℓ at the respective distance, and the number of such labels. The index is a hash table in which keys are edge labels and values are two dimensional matrices. For a label ℓ the matrix contains in position i, j all the nodes n, such that $|W_{n,\ell,i}| = j$, for each $j > 0$.

Note that, once computed for each label ℓ and each $i \leq d$, W compactly represents the neighborhood of a node. For this reason, if we compute W for the nodes of the user sample as well, we can compare nodes in the database and nodes in the user sample, in order to know in advance which nodes can be pruned. We denote the d-neighborhood of a node n_s of graph S by $\mathbb{N}_d^S(n_s)$. A node $n \in N$ of $\mathcal{D} : \langle N, E \rangle$ matches a node $n_s \in N_s$ in the user sample, and therefore is not pruned, if for each label ℓ and a distance $i \leq d$, $|W_{n,\ell,i}| \geq |W_{n_s,\ell,i}|$ (ref. to Theorem 5.9 for a formal proof).

Using the ability to compare nodes through the compact representation of their neighborhood, we devise a way of fast eliminating pairs of the user sample and database

nodes, respectively, that are unlikely to participate in an isomorphism match. Traditional techniques that compute isomorphisms compute matches of the different nodes independently and then try to combine the results. We believe that this process can be optimized further, if the comparison of the nodes takes into consideration the previously computed matches. To implement this idea we exploit the *dual simulation* in Definition 5.6. Note that, in this case, the dual simulation is used to prune nodes in advance and not as a similarity function as in Section 5.5.4.

Deciding whether one graph dual simulates another graph is known to be solvable in polynomial time with respect to the size of the graph [MCF+14]. The main idea of our approach is to perform multiple dual simulations of the user sample on the database graph while pruning the non matching nodes iteratively. The algorithm works as follows. First, it calculates the d-neighborhood for each node of the user sample. Then, a user sample node is selected as starting node. Although any node is a valid starting node we propose to pick the node with the lowest selectivity among the user sample nodes, with the hope to reduce the number of candidate matches between the user sample and database nodes. The selectivity is an estimate of the number of possible matches generated from a user sample node. The idea is to consider the number of adjacent nodes of a user sample node and the frequency of the labels of the edges connected to it. The selectivity of a node n is

$$Sel(n) = \ freq(n) + \sum_{i=1}^{d} \frac{1}{i} \sum_{W_{n,\ell,i}} |E^{\ell}|, \tag{5.1}$$

where the frequency $freq(n)$ of a node n is defined as the sum of the number of outgoing and incoming edges. Similarly, we define the frequency of a label ℓ as the number of edges in the graph having label ℓ and we denote it as $|E^{\ell}|$. The less probable the combination of labels at a certain distance is, the lower the selectivity and the higher is the expected pruning power.

After having selected the starting node n_{min}, the algorithm retrieves the nodes in the database that match the node n_{min} and marks them as candidate mappings $\mu(n_{min})$, where $\mu \subseteq N_S \times N$ is the mapping between user sample and database nodes that the algorithm will compute. Then the algorithm iteratively checks, for each user sample node n_s not yet visited, that each adjacent edge of n_s matches the edges adjacent to

Algorithm 9 IterativePruning

Input: A database $\mathcal{D} : \langle N, E \rangle$
Input: A user sample $S : \langle N_S, E_S \rangle$
Output: A set of candidate mappings $\mu \subseteq N_S \times N$

$\mathbb{N}_d^S \leftarrow d$-neighborhood of S
Vis $\leftarrow \emptyset$ ▷ Visited nodes
$n_{min} \leftarrow \arg\min_{n \in N_S} Sel(n)$
$C \leftarrow \{n_{min}\}$ ▷ Query candidates
$\mu(n_{min}) \leftarrow \{n | \mathbb{N}_d^S(n_{min}) \subseteq \mathbb{N}_d(n)\}$
for each $n_s \in C$ **do**
  if $n_s \xrightarrow{\ell} n'_s \in E_S$ and $n'_s \notin$ Vis **then**
    $\mu(n_s) \leftarrow \mu(n_s) \setminus \{n | n \not\xrightarrow{\ell} n_1, n \in \mu(n_s)\}$
    $\mu(n'_s) \leftarrow \{n_1 | n \xrightarrow{\ell} n_1, n \in \mu(n_s), \mathbb{N}_d^S(n'_s) \subseteq \mathbb{N}_d(n_1)\}$
  else if $n'_s \xrightarrow{\ell} n_s \in E_S$ and $n'_s \notin$ Vis **then**
    $\mu(n_s) \leftarrow \mu(n_s) \setminus \{n | n_1 \not\xrightarrow{\ell} n, n \in \mu(n_s)\}$
    $\mu(n'_s) \leftarrow \{n_1 | n_1 \xrightarrow{\ell} n, n \in \mu(n_s), \mathbb{N}_d^S(n'_s) \subseteq \mathbb{N}_d(n_1)\}$
  $C \leftarrow C \cup \{n'_s | n_s \xrightarrow{\ell} n'_s \vee n_s \xleftarrow{\ell} n'_s\}$
  $C \leftarrow C \setminus \{n_s\}$
  Vis $\leftarrow$ Vis $\cup \{n_s\}$

the nodes $n \in \mu(n_s)$, verifying the label and the direction of the edge. If it does not match, then n is removed from $\mu(n_s)$, otherwise we consider a node n_1 adjacent to n a candidate for the user sample node n'_s adjacent to n_s, i.e., we insert it into $\mu(n_s)$, if the condition described by Theorem 5.9 holds. Finally, the user sample node n_s is marked as visited and removed from the candidate list. The steps of the algorithm are described in pseudo-code in Algorithm 9.

The following theorem guarantees that Algorithm 9 does not falsely discard any node while traversing the user sample nodes. However, it may introduce false positives, i.e., nodes that match the user sample nodes but are not included in an isomorphism.

Theorem 5.9. *Given a database $\mathcal{D} : \langle N, E \rangle$ and a user sample S, let $\mathbb{N}_d$ and $\mathbb{N}_d^S$ be the d-neighborhood of $\mathcal{D}$ and S respectively. If there exists a subgraph-isomorphism $\mu : N_S \to N$, then $\forall n_s \in N_S, \mathbb{N}_d^S(n_s) \subseteq \mathbb{N}_d(n), n \in N, n \in \mu(n_s)$*

Proof. (by contradiction) Suppose that $(n_s, n) \in \mu$, but $\mathbb{N}_d^S(n_s) \not\subseteq \mathbb{N}_d(n)$, then there exists $i, 1 \leq i \leq v$ and a label ℓ such that $|W_{n_s,\ell,i}| > |W_{n,\ell,i}|$. For this reason we can say that there exists $n'_s \in W_{n_s,\ell,i}$, connected to $n''_s \in N_{i-1}(n_s)$ by ℓ, i.e., $n'_s \xrightarrow{\ell} n''_s$. The latter assumption holds since μ is a subgraph-isomorphism. However, there does not exist any $\mu(n'_s) \xrightarrow{\ell} \mu(n''_s)$, which contradicts the subgraph-isomorphism hypothesis. □

Additionally, a guarantee that the algorithm correctly computes multiple simulations of the user sample S, is offered by the following theorem.

Theorem 5.10. *Given a user sample S, if Algorithm 9 terminates with a complete exploration of the nodes S, then there exists in μ a dual simulation $\mathcal{R}$ of S.*

Proof. In order to prove the theorem we need to find at least one subgraph of $\mathcal{D}$ that simulates the sample S. The hypothesis assumes that all the nodes in the sample have been visited, so the set of visited nodes Vis is equal to N_s. We assume, wlog, that initially μ contains all the nodes in the graph. Since the only node in C is n_{min} (Line 4) the algorithm repeatedly inspects the query to find incoming and outgoing edges with some specific label. Given a node $n \in \mu(n_{min})$ and an edge $n \xrightarrow{\ell} n_1$ there are two possibilities: (1) n does not have n_1 such that $n \xrightarrow{\ell} n_1$ or (2) there exists n_1 such that $n \xrightarrow{\ell} n_1$. If (2) holds, either there exists some other node with an ℓ edge, or the algorithm stops with an empty μ. On the other hand, if (1) holds then the algorithm adds n_1 to $\mu(n_1)$ and continues the search. At the end all the nodes and edges of the sample are checked in both directions, incoming and outgoing (Line 7-13), and the dual simulation found in $\mathcal{D}$. □

In the worst case, Algorithm 9 will have to traverse the entire database for each node. Thus, the complexity of the algorithm is $\mathcal{O}(|E| * (|N_S| + |E_S|))$. Since the user sample is typically very small, the algorithm is, for the majority of practical cases, quadratic to the number of nodes. In implementation, to reduce the time computation of μ we used a hash map for storing the nodes of the user sample and their partial mappings.

The set of candidate mappings computed by Algorithm 9 is used to eliminate those nodes of the database that will never participate in an isomorphism with the user sample nodes.

5.5.3.2 An Approximate Solution

In the previous subsection, we described an exact solution to prune the search space, removing nodes that cannot possibly match the user sample. In this subsection, we propose an additional method that removes in advance nodes that are likely to not be relevant for the user, we call this method ApFastXQ.

We aim at restricting in advance the search space in order to search for solutions only in the portion of the graph that is more likely to contain relevant answers. As already mentioned in the previous subsections, both pairs *Yahoo!-Tumblr* and *CBS-Paramount* are part of the solution space, but the pair *Yahoo!-Tumblr* is more relevant to the user, and therefore we would like to restrict our search only to the subgraph that is containing the second but not the first.

In the following, we describe how we model this portion of the graph, which we call *Relevant Neighborhood*. That portion is the subset of nodes with higher proximity to the nodes of the user sample. The intuition behind this is that nodes in the graph that are located far from the user sample will be also semantically distant from the user's intention as expressed in the exemplar query.

We model a *relatedness* measure based on the distance in the graph, and we use it to prune away nodes that are far away from the user sample before even looking for isomorphic structures.

It is clear that, while the approach described in the previous subsection is exact (does not discard any valid answers), this second approach is approximate: some correct answers could potentially be filtered out as they fall out of the *Relevant Neighborhood*. For this reason, we propose a principled way of measuring the *relatedness* and for pruning the graph, aimed at discarding only irrelevant solutions. We implement a function that iteratively retrieves the *Relevant Neighborhood* without traversing the entire graph. As we show later (Section 5.7), thanks to the RelevantNeighborhood algorithm, by operating in this special portion of the graph, we can effectively reduce the search space. The restricted search space can then be given as input to XQ in Algorithm 8, without sacrificing the quality of the results. We can still apply on this subgraph the pruning techniques presented in the previous subsection and then look for isomorphic structures on a much smaller database. Hence, the ApFastXQ algorithm first applies the RelevantNeighborhood algorithm and then FastXQ.

Identifying the Relevant Neighborhood. To find the subset $\mathfrak{Q}_\rho$ of the set of answers $\mathfrak{Q}$, that contains the answers that are relevant to the user, we assume to have some measure of *relevance* ρ. Since the only evidence of the user's intent is the input query Q and the corresponding user sample S that we have found in the first step of the exemplar

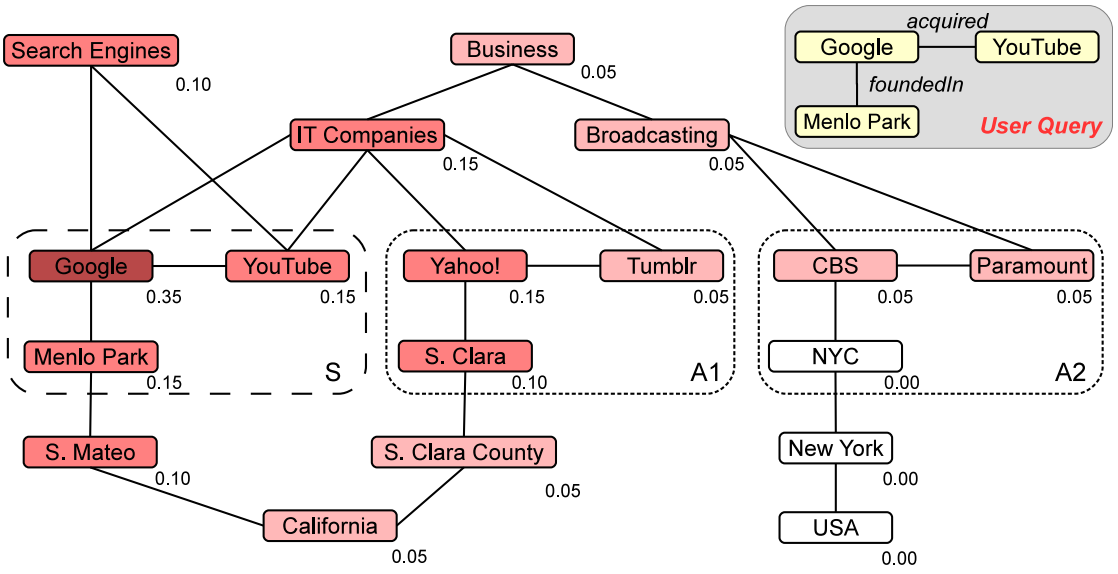


Figure 5.3: A visualization of APPV

query evaluation, we can define the set of relevant answers as $\mathfrak{Q}_\rho = \{A' \in \mathfrak{Q} \mid \rho(A', S) > \tau\}$, with $\tau > 0$ being a minimum threshold.

Since $\mathfrak{Q}_\rho$ is clearly a set of subgraphs in $\mathcal{D}$, we say that the set of solution $\mathfrak{Q}_\rho$ is contained in the subgraph $\mathcal{D}_\rho \subseteq \mathcal{D}$, which is any subgraph of $\mathcal{D}$ that contains all the members of $\mathfrak{Q}_\rho$, relevant answers, and none of the remaining irrelevant solutions in $\mathfrak{Q} \backslash \mathfrak{Q}_\rho$. For this reason we call $\mathcal{D}_\rho$ the *Relevant Neighborhood* of the sample S.

This portion of the graph, being a subgraph itself, is identified by the subset of *relevant nodes* $N_\rho \subseteq N$. Those are nodes whose relevance measure ρ is within the a threshold τ, i.e., $N_\rho = \{n \in N | \rho(n, S) > \tau\}$. Operationally, we first identify the set of relevant nodes N_ρ, and then with only those nodes, we easily construct the subgraph $\mathcal{D}_\rho \subseteq \mathcal{D}$.

In our solution, we implement ρ as a distance measure on the graph, such that it measures the distance of every node from the nodes of the sample N_S, and we keep only nodes that are within a certain distance threshold from the sample. In order to compute this distance we propose the Adaptive Personalized PageRank Vector (APPV), an extension of the Personalized PageRank Vector (PPV), designed to exploit the properties of our problem. This is implemented by the RelevantNeighborhood algorithm. ■

The RelevantNeighborhood Algorithm. Our solution models the computation of the Personalized PageRank vector (PPV) [JW03] which is used as an estimate of the distances of the nodes in the graph from the subset of nodes in the user sample.

Algorithm 10 RelevantNeighborhood

Input: User Sample $S : \langle N_S, E_S \rangle$
Input: Database $\mathcal{D} : \langle N, E \rangle$
Input: Teleportation probability c
Input: Threshold τ
Output: Subgraph $\mathcal{D}' \subseteq \mathcal{D}$

```
1: Ā ← AdjacencyNormalized(D, S)
2: p ← [0] × N
3: for each q_i ∈ N_S do
4:     p[q_i] ← 1/|N_S|
5: v ← ComputeAPPV(Ā, p, c, τ)
6: N_D' ← Nearest(N, v)
7: D' ← GetSubgraph(D, N_D')
8: return D'
```

Personalized PageRank computes the PageRank biased towards the preferences of the user. In our case, user preferences are expressed through the query Q and for this reason we initialize the preference vector according to the nodes in the user sample S, which models the query Q in the database.

The main difference between the original PPV model and our solution, APPV, lays on the semantic of edges. Traditionally, edges between nodes are treated equally as they usually represent just a link from one webpage to another (i.e., they are of the same kind). In contrast, our model adapts to the various edges and their labels, according to S. In particular, the edges in our model may represent different kinds of relationships. It is therefore natural to differentiate based on the information carried by different edges, as some relationships are more informative than others. Moreover, labels that do appear in the user query should be treated differently when computing the PageRank, because they represent the user preference.

Figure 5.3 depicts the output of the computation on the graph of our running example. Here all the nodes have been assigned the weights from the final APPV, computed using the set of nodes in the sample as initial preferences.

The set N_ρ, which satisfies the selectivity requirement, consists of nodes with Personalized PageRank score higher than a minimum threshold τ, $0 < \tau < 1$.

This whole process, presented in Algorithm 10, returns the portion of the graph that is combined with Algorithm 9 to produce a restricted database D' which is provided to Algorithm 8 instead of D.

Assume a model of the database $\mathcal{D} : \langle N, E\rangle$, and let $A^{\mathcal{D}}$ be the adjacency matrix of this graph. If $|N|$ is the number of nodes in the database, then $A^{\mathcal{D}}$ is an $|N| \times |N|$ square matrix. In this matrix, we have that $0 < A^{\mathcal{D}}_{ij} \leq 1$ if and only if the node i has a relationship e^{ℓ}_{ij} with node j with label ℓ; otherwise, we have $A^{\mathcal{D}}_{ij} = 0$. In this way, the element $A^{\mathcal{D}}_{ij}$ models the amount of information that is transferred from node i to node j by the edge e^{ℓ}_{ij} as a function of its label ℓ. In our solution, the values in $A^{\mathcal{D}}$ are proportional to the amount of information [Sha01] carried by the edge e^{ℓ}_{ij}, which is:

$$I(e^{\ell}_{ij}) = I(\ell) = \log \frac{1}{P(\ell)} = -\log P(\ell) \tag{5.2}$$

$$P(\ell) = \frac{|E^{\ell}|}{|E|} \tag{5.3}$$

where E^{ℓ} is the set of edges with label ℓ. Note that the frequency of a label can be easily computed in the database.

In order to account for the importance of the edges in the user sample, we additionally define matrix A^S, which is constructed from the adjacency matrix of the database, but where only entries for edges whose label appears also in S are assigned a non-zero value. In other words, we construct an $|N| \times |N|$ square matrix with $0 < A^S_{ij} \leq 1$ if the nodes i and j are connected by an edge and that edge has a label ℓ that appears as label of one edge in the user sample S, and with A^S_{ij}=0 otherwise.

We then combine the two matrices into the matrix $\bar{A} = A^{\mathcal{D}} + A^S$ and normalize it. Under this transformation $\bar{A}$ becomes the transition probability matrix for the knowledge-base graph, where more relevance is given to edges carrying more information, as well as to edges with labels that appear in the query. We also define $\boldsymbol{p}$, an $|N| \times 1$ column vector, which serves as the normalized preference vector for which $\boldsymbol{p}[i] \neq 0$ *iff* $n_i \in N_S$, i.e., $0 < \boldsymbol{p}[i] \leq 1$ if and only if the node i is in S. Given the column normalized transition probability matrix $\bar{A}$, the teleportation probability c, and the preference vector $\boldsymbol{p}$, our technique adheres to the Personalized PageRank semantics [JW03].

Thus, the APPV $\boldsymbol{v}$ is defined as the stationary distribution of the Markov chain with state transition given by the matrix

$$(1 - c)\bar{A}\boldsymbol{v} + c\boldsymbol{p} \tag{5.4}$$

Algorithm 11 ComputeAPPV

Input: Adjacency Matrix $\bar{A}$
Input: Node vector $\boldsymbol{p}$
Input: restart probability c
Input: threshold τ
Output: Approximate APPV $\boldsymbol{v}$

```
1:  for each q_i ∈ p do
2:      p[q_i] ← p[q_i] × 1/τ
3:  v ← p
4:  while ∃ n_i ∈ p | p[n_i] ≠ 0 do
5:      aux ← [0]
6:      for each n_i ∈ p | p[n_i] ≠ 0 do
7:          particles ← p[n_i] × (1 − c)
8:          for each n_i → n_j ∈ D (Sort by Ā_ij Desc.) do
9:              if particles ≤ τ then
10:                 break
11:             passing ← particles × Ā_ij
12:             if passing ≤ τ then
13:                 passing ← τ
14:             aux[n_j] ← aux[n_j] + passing
15:             particles ← particles − passing
16:     p ← aux
17:     for each n_i ∈ p do
18:         v[n_i] ← v[n_i] + p[n_i]
19: return v
```

where the *teleportation* probability $c \in (0, 1)$ is typically ≈ 0.15, with small changes in this value having little effect in practice [PBMW99].

The exact computation of this vector typically requires $\mathcal{O}(|N|^2)$ time and space. Performing the computation through power iteration requires $\mathcal{O}(|N|t)$ time, where t is the number of iterations to be performed. Nevertheless, this computation is still not practical for very large graphs.

In order to compute this value fast, we extend the template proposed in [BYBC06] and apply an approach similar to the *weighted particle filtering procedure* proposed in [LC10] but extended to correctly take into account the *teleportation probability*, and to consider the non-uniform edge weights that we previously introduced. The extension is shown in Algorithm 11.

Algorithm 11 simulates a set of $1/\tau$ floating particles (line 2) starting from each node with a non-zero value in $\boldsymbol{p}$. At each iteration (lines 6-15), they split among the neighbors of the node they are currently visiting, but we prevent them to split to arbitrarily small

sizes, limiting them to have minimum size τ (lines 12-13). When spreading the particles among the neighbors, the algorithm gives preference to the edges with higher weights. The restart probability c will dissipate part of the particles at every iteration (line 7), and the algorithm will stop when no more particles are floating around.

At the end of the algorithm, we return the APPV containing the scores that have been accumulated through each iteration on every node. We then keep the subset of the graph containing only those nodes with a score higher than some threshold and the edges connected to them (line 6-7 in Algorithm 10). Since we are dealing with an iterative approximation, we keep only those nodes that have been visited by at least one particle, which means that we discard all solutions not greater than τ.

5.5.4 Finding Simulating Answers

In its original formulation, strong simulation is node-label preserving [MCF$^+$14], meaning that the query and the database have labels on the nodes (instead of the edges). On the contrary, our definition is strictly based on edge labels: we require to preserve the relationships among nodes, ignoring the node labels. The adaptation of strong simulation from node-label preserving to edge-label preserving is possible, albeit non-trivial. We discuss the details in the following paragraphs. We also show that it is possible to use the same strong simulation algorithms in our setting. The solution we propose includes the translation of our graph into an *expanded graph*.

Definition 5.11 (Expanded Graph)**.** For a given graph $G : \langle N, E\rangle$ the *expanded graph* is a graph $G^+ : \langle N^+, E^+\rangle$, where each $n_1 \xrightarrow{\ell} n_2, (n_1, n_2) \in E$ is substituted with two edges $n_1 \to n^\ell$ and $n^\ell \to n_2$, where n^ℓ is a new uniquely identified node with label ℓ. The path $n_1 \to n^\ell \to n_2$ is called *expanded edge* and n^ℓ is called *edge-node*.

Clearly, $N^+ = N \cup \{n^\ell \mid \exists\ n_1, n_2 \in N, n_1 \xrightarrow{\ell} n_2\}$ and $E^+ = \{(n_1, n^\ell), (n^\ell, n_2) \mid n_1, n_2 \in N \wedge n_1 \xrightarrow{\ell} n_2\}$.

Figure 5.4 represents an edge $n_1 \xrightarrow{\ell} n_2$ and its expansion. Note that the nodes n_1 and n_2 in the expansion have no labels. We are now ready to prove that the definition of dual simulation in [MCF$^+$14] is equivalent to ours when applied to the expanded graph. Recall that in a node-labeled graph, simulation is defined as follows.

Figure 5.4: An edge (left) and the corresponding expansion (right).

Definition 5.12 (Node-label dual simulation)**.** A node-labeled database $\mathcal{D}_1 : \langle N_1, E_1 \rangle$ dual simulates another graph $\mathcal{D}_2 : \langle N_2, E_2 \rangle$, denoted as $\mathcal{D}_1 \trianglelefteq_D^N \mathcal{D}_2$, if there exists a relation $\mathcal{R}$, such that, for every node $n_1 \in N_1$ and $n_2 \in N_2$ for which $(n_1, n_2) \in \mathcal{R}$: (1) for all $n_1 \xrightarrow{\ell} n_1'$, exists n_2' such that $n_2 \to n_2'$ and $(n_1', n_2') \in \mathcal{R}$, (2) for all $n_1'' \xrightarrow{\ell} n_1$, exists n_2'' such that $n_2'' \xrightarrow{\ell} n_2$ and $(n_1'', n_2'') \in \mathcal{R}$.

We need to prove that edge-label strong simulation is equivalent to node-label simulation on expanded graphs. We first prove the following lemmas.

Lemma 5.13. *Given two databases $S : \langle N_s, E_s \rangle$ and $\mathcal{D} : \langle N, E \rangle$, $S \trianglelefteq_D \mathcal{D} \Leftrightarrow S^+ \trianglelefteq_D^N \mathcal{D}^+$.*

Proof. The structure of the proof is as follows. We prove both arrows separately constructing another dual simulation starting from the one existing by hypothesis.

($\Rightarrow$): Given a dual simulation relation $\mathcal{R}_D$ from s to $\mathcal{D}$ we construct a relation

$$\mathcal{R}'_D = \mathcal{R}_D \cup \mathcal{R}_D^+,$$

where $\mathcal{R}_D^+ = \{(s^\ell, n^\ell) \mid s^\ell \in N_s^+, n^\ell \in N^+, (s_1, n_1), (s_2, n_2) \in \mathcal{R}_D \wedge s_1 \xrightarrow{\ell} s_2, n_1 \xrightarrow{\ell} n_2\}$. Note that for the generality of s_1, s_2, n_1, n_2, $\mathcal{R}_D^+$ contains edges in both directions. $\mathcal{R}'_D$ is, in fact, a dual simulation from S^+ to $\mathcal{D}^+$. Suppose $\mathcal{R}'_D$ is not a dual simulation, then it must exists $s \in N_s^+$ such that it for any $n \in N^+$, $(s, n) \notin \mathcal{R}_D$. We have two cases:

(1) $s \in N$. This is a contradiction, since $\mathcal{R}_D$ is a dual simulation it exists a n, such that $(s, n) \in \mathcal{R}_D$.

(2) $s \in N^+ \setminus N$. This means that s is an edge-node and exists a label ℓ and $s_1, s_2 \in N_s$, such that $s_1 \xrightarrow{\ell} s_2$. By hypothesis exists $n_1, n_2 \in N$ such that $(s_1, n_1) \in \mathcal{R}_D$, and $(s_2, n_2) \in \mathcal{R}_D$. However, in the expanded graph $n_1 \to n^\ell \to n_2$, implying that $(s^\ell, n^\ell) \in \mathcal{R}'_D$ contradicting the hypothesis.

Algorithm 12 StrongSimSearch

Input: User database $\mathcal{D}$: $\langle N, E\rangle$
Input: User sample S
Output: Set of simulating answers $\mathfrak{Q}$
1: $\mathcal{D}^+ \leftarrow$ Expand($\mathcal{D}$)
2: $S^+ \leftarrow$ Expand(S)
3: $\mathfrak{Q} \leftarrow \emptyset$
4: $\mathfrak{Q}^+ \leftarrow \mathsf{Match}(\mathcal{D}^+, S^+)$ ▷ Algorithm from [MCF+14]
5: **for all** $\mathfrak{q}^+ \in \mathfrak{Q}^+$ **do**
6: $\quad \mathfrak{Q} \leftarrow \mathfrak{Q} \cup$ Contract($\mathfrak{q}^+$)

($\Leftarrow$): The proof is similar to the forward arrow, noticing that $\mathcal{R}_D = \mathcal{R}'_D \setminus \mathcal{R}^+_D$, and will be omitted. □

We are now ready to prove the following Theorem.

Theorem 5.14. *Let $S : \langle N_s, E_s\rangle$ and $\mathcal{D} : \langle N, E\rangle$ two databases, $S \trianglelefteq_S \mathcal{D} \Leftrightarrow S^+ \trianglelefteq^N_S \mathcal{D}^+$.*

Proof. Recall that from Definition 5.8, two graphs are strongly similar if there exists a node $n \in N$ and a $d-$graph $\mathcal{D}[n, d]$ such that (1) d is equal to the diameter of S, and (2) $S \trianglelefteq_D \mathcal{D}[n, d]$ with the maximal dual simulation. If $S \trianglelefteq_S \mathcal{D}$, by Lemma 5.13 follows that $S^+ \trianglelefteq^N_D \mathcal{D}^+[n, d]$ and it easy to see that d is the diameter of S^+. It also flows that $S^+ \trianglelefteq^N_D \mathcal{D}^+[n, d]$ with the maximal relation, since the relation, as defined in Lemma 5.13 contains all the pairs plus the pairs included in the expanded edges. □

Theorem 5.14 states that it is sufficient to run the Match algorithm from [MCF+14] on an expanded graph, and then remove all the edge-nodes and the matching from the expanded graph to obtain valid strong-simulating results for the original graph. Algorithm 12 shows the pseudocode of the strong-simulation algorithm. First, the graphs S and $\mathcal{D}$ are expanded using the procedure Expand (Line 1,2); then, the Match algorithm is used to find strong-simulating answers $\mathfrak{Q}^+$ in the expanded graphs S^+ and $\mathcal{D}^+$ (line 4). Finally, all the results in $\mathfrak{Q}^+$ are contracted using the Contract function to remove the expanded-edges (line 5-7).

Algorithm complexity. The Match algorithm on $\mathcal{D}^+$ runs in $\mathcal{O}(|N^+|(|N^+| + (|N^+_s| + |E^+_s|)(|N^+| + |E^+|)))$ as shown in [MCF+14]. Since the size of the user sample is small, $|N^+_s|$ is bounded by a small constant and we can consider Match to run in $\mathcal{O}(|N^+|(|N^+| + (|N^+| + |E^+|)))$. On the other hand, $|N^+| = |N| + |E|$ since for each edge we add a

node and $|E^+| = 2|E|$. Therefore, since both EXPAND and CONTRACT execute in $\mathcal{O}(|E|)$ time, the overall complexity is dominated by Match, which runs in $\mathcal{O}(|N|^4)$ for expanded graphs.

Applying pruning techniques. Algorithm 12 works with any kind of graph, but expanding the entire database may be time consuming. A legitimate question then is whether ITERATIVEPRUNING can be applied with simulation.

Indeed, this is possible by relaxing the constraint in Theorem 5.9. It easily follows from Theorem 5.9 and Definition 5.6 that in a database $\mathcal{D} : \langle N, E \rangle$, a node $n \in N$ matches a node $n_s \in N_s$ in the user sample S, so it is not pruned, if for each label ℓ and a distance $i \leq d$, if $|W_{n_s,\ell,i}| > 0$ then $|W_{n,\ell,i}| > 0$. We refer to this algorithm as ITERATIVEPRUNING*.

We call FASTXQSIM the algorithm that derives from XQ, when instantiated with the STRONGSIMSEARCH similarity function described in Algorithm 12 and the pruning algorithm ITERATIVEPRUNING*.

Regarding the restriction of the search space presented in Section 5.5.3.2, we note that no changes are needed to RELEVANTNEIGHBORHOOD. Therefore, for the case of strong simulation, we can use the APFASTXQ algorithm by first applying the RELEVANTNEIGHBORHOOD algorithm presented above, and then the FASTXQ with ITERATIVEPRUNING*. We refer to this new algorithm as APFASTXQSIM.

5.6 Ranking Query Answers

Once the answers have been computed from the user sample, they need to be ranked in order to either be returned sorted to the user that posed the query, or to select only the k most promising candidates, i.e., the top-k. To do this, we introduce a novel ranking function that is a linear combination of two scores, namely, the structural similarity score $\mathcal{S}$ based on the d-neighborhood and the amount of information as provided by the Personalized PageRank, which indicates the importance of a label in the graph. The score of each answer is computed by using the above two parameters to compare the answer to the user sample.

Most node similarity measures proposed in the literature are based on the concept of graph similarity and isomorphism. This is the case for Graph Edit Distance [GXTL10], which is computed with a reduction to graph isomorphism, and is therefore inapplicable to our problem, due to its high time complexity. A different method is proposed in [KLY+11] and is based on a vectorial representation of nodes. This idea seems suitable for our settings, thus we extended it in order to capture the differences among nodes that emerge when taking into account the edge-labels of the neighbors. We also embed distance information aiming at giving different weights to nodes based on their distance from the sample. Thus, for every node n we build a vector containing a value for every label $\ell \in L$ in the graph, and we compute this score as

$$\sigma(n, \ell) = \sum_{i=1}^{d} \frac{I(\ell)|W_{n,\ell,i}|}{i^2}$$

Given the vectorial representation of two nodes, we compute the node similarity $\mathcal{S}$ using a metric for vectors, such as the Jaccard, euclidean distance or cosine similarity. Note that our vectorial representation contains already the computed score σ. In our experiments we use cosine similarity but any other similarity metrics can also be used. Therefore, the *structural similarity* between a node n_s of the user sample and any matching node n is computed as follows:

$$\mathcal{S}(n_s, n) = \frac{\sum_{\ell \in \mathcal{L}} \sigma(n_s, \ell)\sigma(n, \ell)}{\sqrt{\sum_{\ell \in \mathcal{L}} \sigma(n_s, \ell)^2} \sqrt{\sum_{\ell \in \mathcal{L}} \sigma(n, \ell)^2}}$$

The structural similarity above does not take into account the proximity measure of the results with respect to the user sample. Therefore, we consider a linear combination, parametrized by λ, between the node similarity (structural) and the Personalized PageRank (proximity) as follows.

$$\rho(n_s, n) = \lambda \mathcal{S}(n_s, n) + (1 - \lambda)\boldsymbol{v}[n] \tag{5.5}$$

where $\boldsymbol{v}[n]$ is the APPV defined in Section 5.5.3.2.

This score is then averaged over the number of nodes matching the user sample nodes in the similarity relation $\mathcal{R}$ and summed to the individual score for each node in the sample.

$$\rho(S, \mathcal{R}) = \sum_{n_s \in N_s} \left(\frac{\sum_{n \in \mathcal{R}(n_s)} \rho(n_s, n)}{|\mathcal{R}(n_s)|} \right) \tag{5.6}$$

Note that the choice of λ in Equation 5.5 is data dependent. A value λ close to 1 favors results that are mostly similar to the neighbors of the user sample nodes. On the other hand, a value close to 0 will take into account only solutions that are close to the original query. For this reason, we can see λ as a diversification parameter that depends on the user and on the data. This is also the approach taken by most diversification models [AGHI09].

5.7 Experimental Evaluation

In this section, we experimentally validate our solution by comparing it to other approaches, and measuring its performance. We discuss differences between isomorphism and strong simulation.

Queries: We extracted 100 real queries from the AOL query log, and manually mapped them to the knowledge base[3]. These queries are highly heterogeneous in terms of size and frequency of edge labels. Each query has 2 to 11 edges and diameter up to 8, while the average diameter is 2.8, in line with any real world bulk of queries [GFMPdlF11]. In order to test our algorithms with different query shapes, a query can contain cycles, paths, trees or complete graphs.

Datasets: We downloaded the full Freebase knowledge-base [Goo14] in April 2014, obtaining a connected graph of 76M nodes and 314M edges, with about 4.5K distinct edge types. We refer to this dataset as `Real`. To the best of our knowledge this is the biggest graph used in this context in the literature, and the first time that the entire Freebase graph is used for this purpose. While related works [KLY+11, YYH04, WDT+12] use a small part of Freebase, we explored solutions that scale to its full size.

[3]List of queries: `http://www.mi.parisdescartes.fr/~themisp/exemplarquery-ext/`

Based on `Real` we generated 10 synthetic datasets, embedding 20 samples of the test set in different points of the graph: we performed a breadth first traversal from a fixed starting node and randomly chose to embed an answer according to a distribution that decreases exponentially with the distance from the starting node (thus modeling answers at varying distances). For the scalability tests, we generated graphs having 0.5M, 1M, 5M, 10M and 20M nodes, and 1K embedded queries. We denote them as `GSize-x`, where `x` is the graph size. Similarly, we generated graphs with 10M nodes, and 0.5K, 1K, 2K, 5K and 10K embedded answers. We denote them as `QSize-x`, where `x` is the number of generated answers.

Experimental Setup: We experimentally evaluate the impact of the parameter d on the running time (see Section 5.7.3). We notice that $d = 2$ is a good choice since any larger value does not significantly reduce the time, while requiring more space to store the d-neighborhood. We observe that $\lambda = 0.3$ (see Section 5.6) is a good compromise for retrieving diverse and qualitative results. In Section 5.7.4, we study the effect of varying the threshold parameter τ (see Section 5.5.3.2), for which the default value is 0.003. All the reported results are averages over 5 consecutive runs. We implemented our solution in Java 1.8, and ran the experiments on a $i686$ Intel Xeon $E52440$ 2.40GH machine with 12 Cores in hyper-treading and 188Gb RAM, over Linux kernel $v3.13.0$. The graphs are loaded into main memory using our graph library available under open source license[4].

Implemented Algorithms: Apart from FastXQ, ApFastXQ, FastXQSim and ApFastXQSim, we implemented three additional algorithms from related works:

QueryReformulation: An algorithm that produces query reformulations by mining sessions from query logs in a term-level fashion [WZ08]. The model is trained on the AOL query log and the suggestions are based on our query test set.

EQ-Graph: Entity-query graph is a model that computes serendipitous suggestions starting from entity mentions in a page [BDFMWB13]. For our queries to work in this setting, we associated to each node the corresponding Wikipedia page (or the best Wikipedia

[4] `https://github.com/mutandon/Grava`

page that represents the node). The model is trained on a big query log from the Yahoo! Search Engine.

NeMa: This algorithm [KWAY13], and other previous works, are based on the assumption that there exists a truly small set of correct answers to a graph query, which is not true in our case. Therefore, we implemented their technique taking into account edge label matches instead of node matches. The authors kindly provided us a C++ implementation (compiled using gcc *v*4.4.3).

Summary of Results: Our user studies demonstrate that 92% of the users exemplar queries are relevant and useful for search tasks, and that existing approaches are not able to provide effective solutions for our problem, while our method identifies meaningful results with 81% precision. We observe that the IterativePruning Algorithm saves on average 30% time resulting in a pruned graph 80% smaller, with even higher improvements when we choose starting nodes with low selectivity. More than 50% of the queries take less than 1 second for $\tau \geq 0.003$. With smaller τ a small portion of queries takes more than 10s. The set of results measuring performance demonstrate the scalability of our approach to the largest knowledge-graph available in the field (76M nodes, 312M edges), while maintaining interactive response times. Finally, we witness that although strong simulation runs slower than isomorphism, it retrieves 34% more nodes (i.e., entities).

5.7.1 Usefulness

In order to assess the quality of the proposed solution, we conducted the following user study. We asked 94 users (uniformly distributed with respect to education level, age and country) to evaluate our system. For each query in the test set, we provided an explanation of the topic, the query intention, and our answer set with the top-10 results provided by our ranking function. We asked each user to rate each result as *irrelevant*, *weakly related*, or *very related* with respect to the topic and the expressed query intention. Each user evaluated between 2 to 10 queries (on average 8). The users provided 4540 marks in total, shown in Figure 5.5: 81% of our results are marked as relevant (weakly or strongly) and only 19% of them are not considered relevant suggestions. Out of the 427 suggestions we produced, 172 (i.e., 40%) are judged highly relevant by more than 50% of

the users. Note also that each exemplar query contains at least one relevant (weakly or strongly) result for 99% of the users. Moreover, we ask each user to express her opinion with respect to (a) the idea of using examples as a search paradigm, (b) whether she already had the need of searching using exemplar queries, and (c) the usefulness of the system in general. As shown in Figure 5.6, 92% of the users considers the exemplar queries paradigm and the overall system useful for retrieving additional and relevant information. Moreover, 62% of the people interviewed declared that they had already had the need to perform this kind of exemplar queries search in the past (but there was no system to support them).

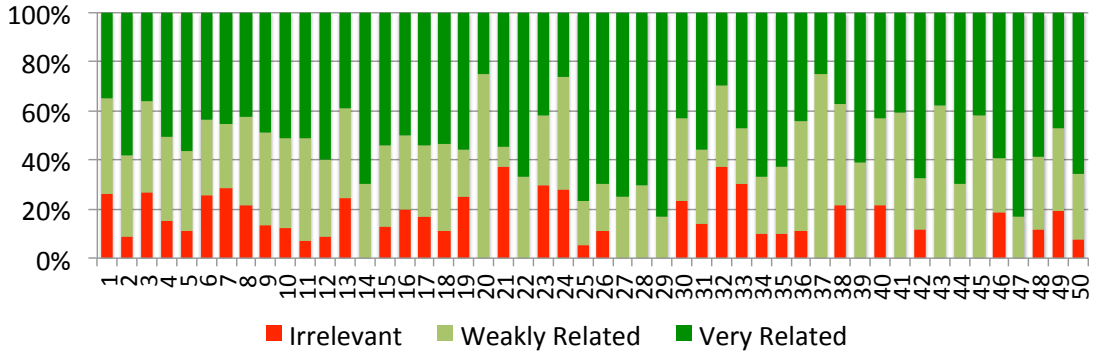


Figure 5.5: Percentage of relevant and irrelevant results per query

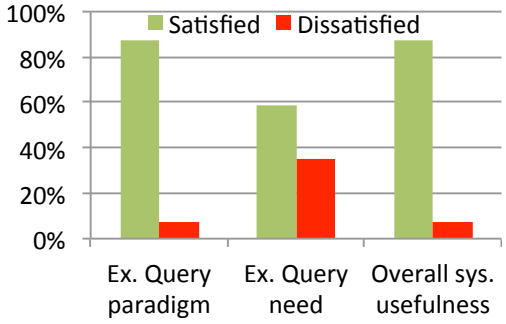


Figure 5.6: Percentage of satisfied and dissatisfied users

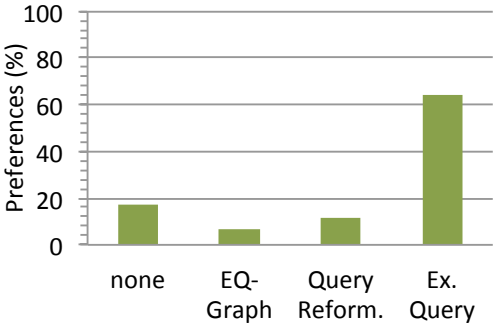


Figure 5.7: Comparison of methods on Exemplar Query task

5.7.2 Comparison to Previous Work

In the following we compare our method against two different approaches: (a) algorithms that produce related queries, and (b) an approximate query answering technique for graphs.

Related Queries: We implemented and compared with the methods QueryReformulation and EQ-Graph mentioned earlier, through a user study similar to the one presented in Section 5.7.1. For each query in the test set, we presented to users three groups of suggestions: one produced with our method, and one produced by each one of the two methods above. We then asked users which of the three groups of suggestions they considered the most helpful for each query task.

The results, depicted in Figure 5.7, show that in 64% of the cases the users preferred our solution to the other two. Furthermore, for 78% of the queries that received more than 2 marks, the majority of users preferred our solution. In 18% of the cases none of the proposed solutions were satisfying, neither the answers proposed by our model nor those produced by the other algorithms. Overall, the two competing approaches together was preferred by less than 30% of the users, none of them choosing the two approaches in all the queries.

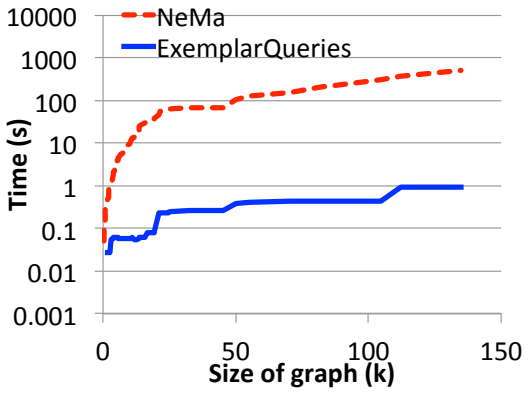


Figure 5.8: Time vs Size of graph with NeMa and our approach (`Real` dataset).

Approximate Query Answering on Graphs: We now present the comparison between our approach and NeMa [KWAY13], a state of the art technique for answering approximate queries on graphs. Since on `Real` a single query takes NeMa more than 13 hours to process, we test NeMa on graphs obtained after applying RelevantNeighborhood on our query test set, thus giving it an advantage. The results in Figure 5.8 show that NeMa is almost three orders of magnitude slower than our algorithm. This suggests that a query answering technique for graphs is not applicable to our setting.

We also provide anecdotal evidence comparing the top-5 results from our method and NeMa. Tables 5.1a and 5.2a show the top-5 results of NeMa for two different exemplar queries compared with the results of our algorithm, shown in Tables 5.1b and 5.2b (for

Google - YouTube - Menlo Park
Yahoo! - LAUNCH Media - Stanford Univ.
Yahoo! - Musicmatch - Stanford Univ.
Yahoo! - Right Media - Stanford Univ.
Yahoo! - Inktomi Corporation - Stanford Univ.

(a) Top-5 results for NeMa.

Google - AdMob - Menlo Park
Google - DoubleClick - Menlo Park
Yahoo! - del.icio.us - Santa Clara
Microsoft - Powerset - Albuquerque
A&E Television - Lifetime Ent. Services

(b) Results for exemplar query with isomorphism.

Google - AdMob - Youtube [...] - Menlo Park
YouTube - Next New Networks - San Mateo
Yahoo! - Inktomi - Del.icio.us, Inc. [...] - Santa Clara
AOL - Sphere - Netscape - USA
John Wiley & Sons - InfoPOEMs - New York City

(c) Results for exemplar query with strong simulation.

Table 5.1: Comparison of results for query "Google - YouTube - Menlo Park".

our algorithm, we report the top-2 results containing query terms, the top-2 results not containing query terms, and for reference, the lowest ranking result). We observe that if the structure of the exemplar query is complex (e.g., it contains cycles), NeMa fails to find the correct answers, mapping different query nodes on the same graph node as depicted in Table 5.2a-row 3, where the same graph node,"Oral Transmission", is used twice. Actually, 87% of the answers produced by NeMa are not isomorphic to the test queries, producing results that contain the same node more than once, and thus, leading to poor results. Furthermore, the top answers proposed by NeMa for Q2 contain diseases that are not sexually transmitted (e.g., diabetes that is ranked 2nd), a situation that does not occur with our algorithm. For strong simulation, Tables 5.1c and 5.2c report the top-2 results containing query terms, the top-2 results not containing query terms and the result with the lowest ranking score. Note that, with strong simulation, some answers include more nodes than those in the query. For instance, the first result in Table 5.1c lists all the acquisitions made by Google[5] and finally, Menlo Park. The maximality enforced by strong simulation compresses several isomorphic answers in one single answer. Similarly, the third result (that does not contain any node of the query),

[5]For sake of presentation we do not report the complete list of entities in the answer.

Water purification - Fecal-oral route - Cholera
Smoking cessation - Vector - Diabetes mellitus
Oral Transm. - Cytomegalovirus - Oral Transm.
Oral Transm. - Cerebral palsy - Cytomegalovirus
Water purification - Fecal-oral route - Cholera

(a) Top-5 results for NeMa.

Sex - HIV infection - Safe sex
Sex - HIV infection - Sexual abstinence
Safe sex - Vertical transmission - Hepatitis B
Safe sex - Vertical transmission - Syphilis
Hand washing - Droplet Contact - Cold

(b) Results for exemplar query with isomorphism.

Sex - Condom - HIV infection - Safe Sex [...] - Candidiasis
Sex - Condom - Unsafe Sex - [...] - Pelvic inflammation
Vaccine - Poor Hygiene - Immunodeficiency [...] - Influenza
Contact with infected person - Aciclovir [...] - Chickenpox
DPT vaccine - Child age - Droplet Contact [...] - Pertussis

(c) Results for exemplar query with strong simulation.

Table 5.2: Comparison of results for query "Condom - Sex - HIV infection".

represents all the acquisitions by Yahoo and the Santa Clara node. As opposed to isomorphism, strong simulation correctly groups answers having the same root node (e.g. Yahoo). Table 5.2c shows how the maximality condition bounds the size of the answers. Indeed, all the results that are in the first two rows of Table 5.2b are condensed in the first result in Table 5.2c that represents all the risk factors and prevention methods for sexually transmitted infections. We observe that some nodes are still repeated among different results, concluding that strong simulation does not necessarily collapse all the redundant information in one single answer. Nonetheless, with an appropriate presentation of the results we already obtain good clusters of answers. Note that Table 5.2c-rows 3,4 presents relevant answers about other contagious infections not related to sex. These results are also present in the isomorphic answers in low ranked position.

5.7.3 Pruning Effectiveness

In the next experiments we study the impact of pruning on query time and the effect of selectivity on pruning time.

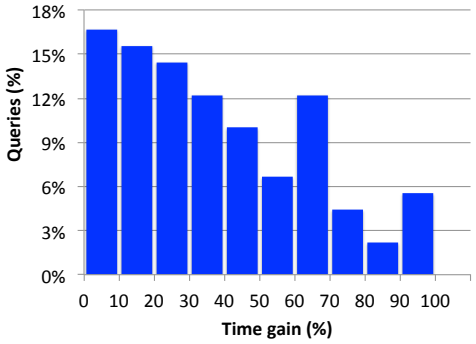


Figure 5.9: Execution time gain distribution as a result of pruning (`Real` dataset).

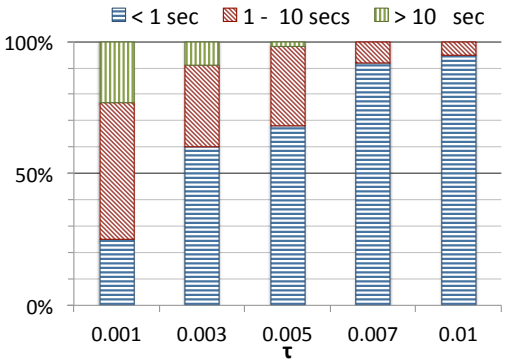


Figure 5.10: Distribution of Running Time vs ApFastXQ threshold (`Real` dataset).

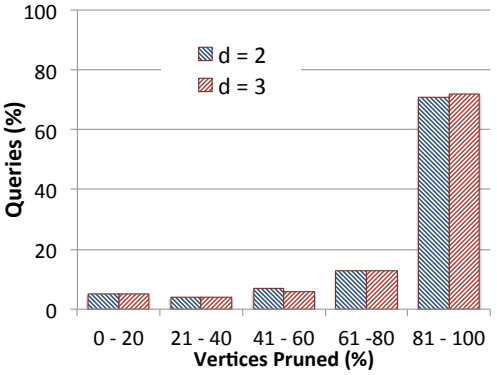


Figure 5.11: Pruned Edges Distribution (`Real` dataset).

Pruning impact: We perform a batch of experiments using repeatedly the query test set, comparing the query time with and without applying IterativePruning, and depict the results in Figure 5.9. The parameter d of IterativePruning (Section 5.5.3.1) determines how large is the representation of the neighborhood on each knowledge-graph node and it is compared to the similar representation for each node of the query. By definition, a higher value of d causes a more aggressive pruning of the search space. (We note that, as discussed in Section 5.5.3.1, our pruning technique does not modify the quality of the final result set, neither does it discard any relevant result.) Nonetheless, any value for d larger than the query diameter has no impact on pruning power as highlighted in Theorem 5.9. Figure 5.11 validates the latter claim showing that the benefit with $d = 3$ minimal. This result shows that, in general, the graph structures are not captured by the $d-$neighborhood of a node for edges at distance 3 or more.

Overall, applying pruning results in 3% to 99% less query time. Furthermore, we observe that for 17% of the queries, pruning does not affect query time. We notice that pruning is more effective when the frequencies of the edge labels of the sample in the graph are high, since a large part of the graph is eliminated with fewer operations. This observation allows us to run the IterativePruning on demand. On average, IterativePruning reduces query time by 30% and the graph size by 80% (by removing non-matching edges). This entire batch of experiments takes 38 minutes to execute without pruning and 17 minutes with pruning, saving 55% of the total time.

Pruning Selectivity: We study the performance of pruning in terms of time as a function of the selectivity of the starting node in the sample. Remember that low selectivity means better pruning (see Equation 5.1). We run experiments measuring the correlation between time and selectivity, selecting the different nodes of the sample as starting nodes. The results show a positive correlation of 0.57 between selectivity and time performance, which is statistically significant at the 0.01 significance level. We conclude that starting from a low selective node positively impacts the pruning time, with savings up to 87%.

5.7.4 Calibrating RelevantNeighborhood

We study the effect of τ on RelevantNeighborhood in terms of time and quality of the results. Parameter τ of RelevantNeighborhood determines the degree of approximation of the estimation of PPV of each node and is directly related to the number of answers retrieved and to the running time. In Figure 5.16a and 5.16b, we plot the size of the neighborhoods (counts of vertices and edges from the graph) visited for increasing values of τ (from 0.001 to 0.01), and the number of answers retrieved in each case. We refer to visited vertices/edges as the vertices/edges retrieved by our RelevantNeighborhood algorithm. Relevant answers are found in the graph containing only such nodes and edges.

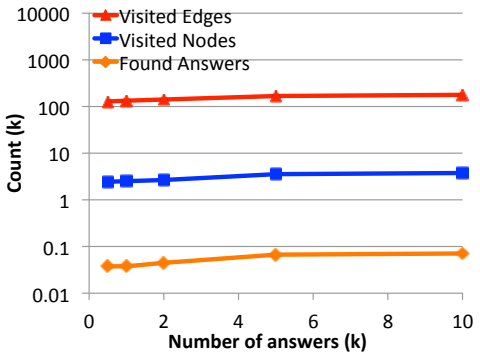
We witness an exponential decay in the number of visited nodes and edges as τ increases, which is directly proportional to the number of answers retrieved. This is mirrored, in Figure 5.14a and 5.14b by a decrease of the running time needed to retrieve the relevant neighborhood and to prune it. In line with this, Figure 5.15a and Figure 5.15b show that with larger values of τ the total time needed to compute the results decreases

in the same manner. Thus, with larger neighborhood we expect to find, as we do, more answers to the query, but they will also require more time to be computed. We experience a discrepancy in the number of answers retrieved with isomorphism and with strong simulation. This discrepancy apparently favors isomorphism, but this is not the correct analysis. Conversely, answer graphs retrieved with strong simulation are usually 34% larger than graphs retrieved with isomorphism due to the maximality property of strong simulation. Indeed, a node in the user sample could be matched to multiple nodes at the same time, ideally collapsing multiple distinct isomorphic answers. We expand over this issue in Section 5.7.6

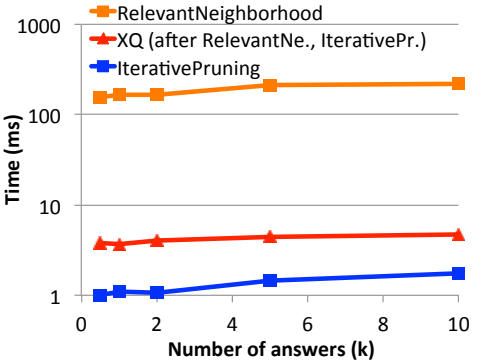
τ	$P@1$	$P@5$	$P@10$	$P@50$	$P@100$
0.002	1	0.99	0.99	0.85	0.75
0.003	1	0.97	0.94	0.80	0.73
0.004	1	0.95	0.93	0.71	0.60
0.005	1	0.94	0.92	0.66	0.56

Table 5.3: Precision of ApFastXQ varying τ

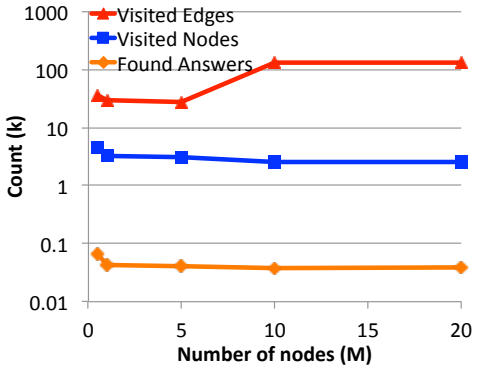
We now evaluate the quality of the answers produced by ApFastXQ, by measuring precision at *1,5,10, 50,100*, where precision at k (abbreviated $P@k$) is defined as the fraction of results produced by FastXQ that are also produced by ApFastXQ in the first k positions. Table 5.3 shows that overall precision is high, especially for the top positions. Any value of τ between 0.003 and 0.005 is a reasonable choice, leading to high precision and an average query time of less than 2.4 seconds. Evidently, the choice of this parameter depends on the application. In a biological setting, where precision is more important than time, $\tau = 0.002$ could be a reasonable choice, producing very precise answers in about 10 seconds. On the web, where timely answers are needed, $\tau = 0.005$ can still offer precise answers in the top positions, in less than 1 second. In our experiments, we use $\tau = 0.003$. When analyzing the performance of our approach, we measure the correlation between the search time with isomorphism and a number of query characteristics, namely diameter, density, number of repeated edge labels, and average label frequency. With isomorphism queries featuring a large number of repeated edge labels positively correlates with the running time and the correlation is statistically significant with p-value < 0.001. The same observation does not hold for strong simulation. We observe a weak correlation, with p-value < 0.01, between search time and the average number of times a label appears in the query.



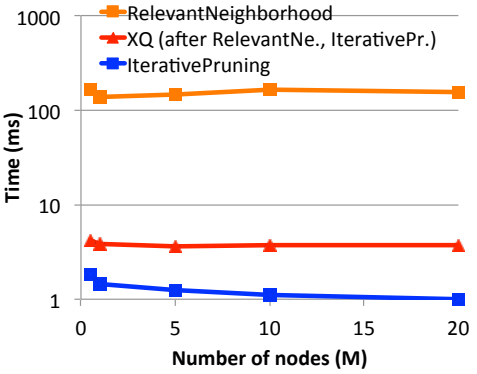
(a) Count vs number of answers (`QSize-x` dataset).



(b) Time vs number of answers (`QSize-x` dataset).



(c) Count vs number of nodes (`GSize-x` dataset).



(d) Time vs number of nodes (`GSize-x` dataset).

Figure 5.12: Scalability experiments varying number of answers and number of nodes in the graph.

5.7.5 Scalability

We present the scalability experiments as a function of the number of answers and the size of the database. In Figure 5.12a we show the number of visited edges and nodes, as well as the number of results when the number of embedded answers increase (recall that `QSize-x` contains exactly x answers for each exemplar query). The time of ApFastXQ is the sum of the times shown in Figure 5.12a. We observe that using RelevantNeighborhood as the number of answers increases from 60 to 100, the number of explored nodes remains almost the same. This behavior is expected, since RelevantNeighborhood does not explore more nodes as long as the structure of the graph remains almost unchanged, but it finds more answers in the same subgraph.

Conversely, if the size of the dataset increases and the number of answers is fixed (Figures 5.12c and 5.12d) it is less likely to find answers close to the exemplar query. As expected, since the number of nodes explored is almost the same (see Figure 5.12c) the

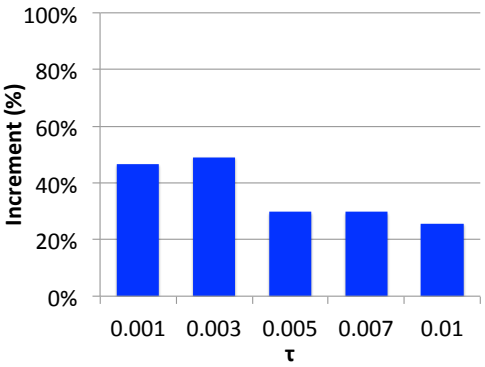


Figure 5.13: Difference in Cardinality of answer vertices (`Real` dataset).

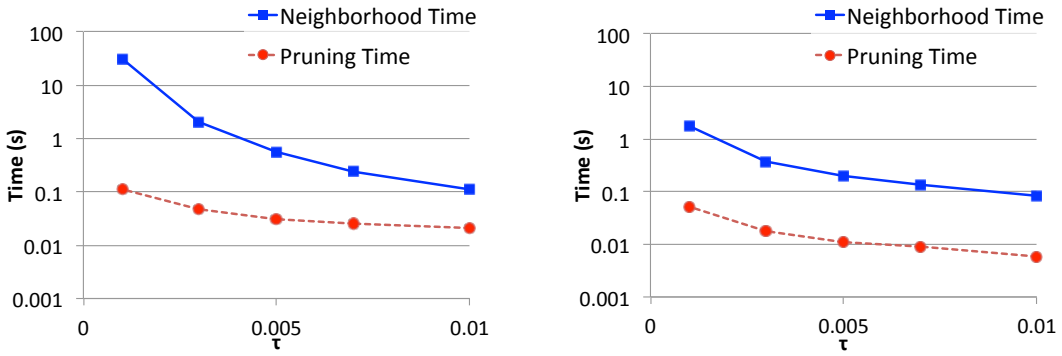
time is constant.This supports our design choice, since changes in the peripheral part of the graph do not affect the APPV algorithm.

5.7.6 Remarks on Isomorphism and Simulation

We discuss the principal differences between ApFastXQ and ApFastXQSim. Recall that ApFastXQ differs from ApFastXQSim only in the similarity function adopted. Consequently, the time required by the RelevantNeighborhood and IterativePruning algorithms is always the same, hence it is reported only once in Figure 5.14a and 5.14b. Therefore, the running time in Figure 5.15a is affected by the difference in computation between the two algorithms. In particular, we notice that the time required to compute the set of maximal d-graphs is much higher than that for isomorphism for very few large and complex queries. This becomes particularly evident in Figure 5.15b that shows roughly the same median running time for the two approaches.

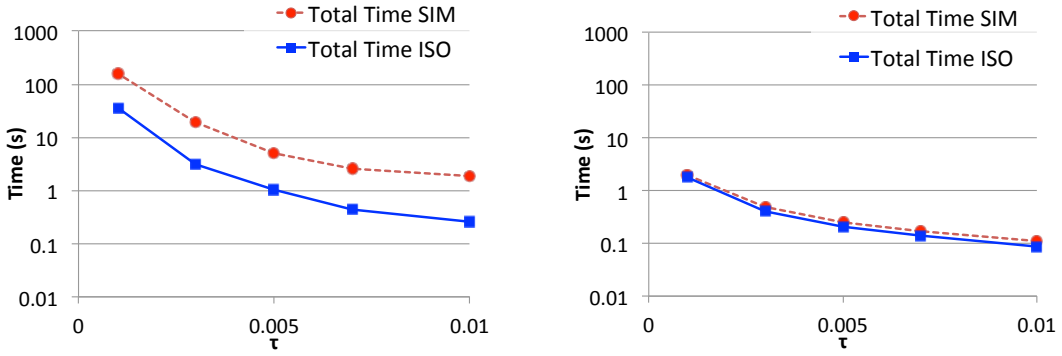
As previously noticed, there is a big difference in the number of answers retrieved (see Figure 5.16a and 5.16b). This is not only a natural consequence of the strong simulation, but is also a desirable effect, because all the results are simply grouped in more large answers. Figure 5.13 validates this claim, showing that strong simulation retrieves from 22% ($\tau = 0.01$) to 48% ($\tau = 0.0003$) more nodes than isomorphism. The figure reports the average increment as a function of τ and is obtained by counting the difference in the number of nodes in both answer sets. As expected, the answer set retrieved with strong simulation is a superset of the one retrieved by isomorphism. We conclude that answering exemplar queries with either strong simulation or isomorphism is equivalent in terms of performance and that strong simulation captures more answers with an acceptable slack

in terms of matching quality. Hence, the choice of the similarity function depends on the user requirements, but, in both cases, the semantics of the query is preserved.



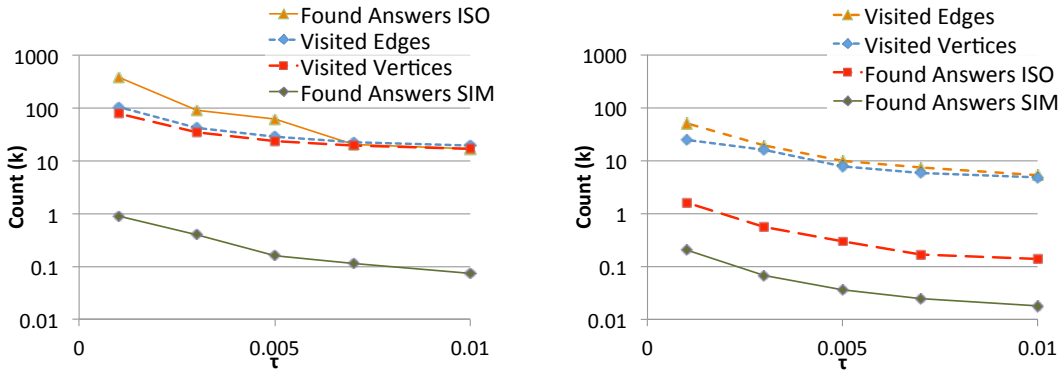
(a) Average neighbor and pruning time. **(b) Median** neighbor and pruning time.

Figure 5.14: Study of the RelevantNeighborhood and IterativePruning time as a function of the threshold τ, comparing Isomorphism and Simulation with the ApFastXQ/ApFastXQSim Algorithms.



(a) Average total time. **(b) Median** total time.

Figure 5.15: Study of the **total** time as a function of the threshold τ, comparing Isomorphism and Simulation with the ApFastXQ/ApFastXQSim Algorithms.



(a) Average count in terms of number of answers and visited edges/nodes. **(b) Median** count in terms of number of answers and visited edges/nodes.

Figure 5.16: Study of the counts as a function of the threshold τ, comparing Isomorphism and Simulation with the ApFastXQ/ApFastXQSim Algorithms.

Chapter 6

Conclusions

Query answering is the process of retrieving objects from a database satisfying a set of conditions expressed in a query. Since the number of users accessing data in any format, either online or in standalone applications, is constantly increasing the design of search systems has to reflect the user needs. This becomes critical with novice users, i.e., users with little or no experience with computers, because they require far more support from the system than practiced users. Novice users are difficult customers since they use a restrict vocabulary when formulating the query, they require correct and timely answers, and they need a direction from the system.

For these challenges this dissertation initiated a study. As such, we proposed methods to cope with very common problems in query answering: (1) the empty answer problem, in which the user provides too restrictive queries and the system returns no answers; (2) the information overload problem, which deals with the opposite problem of having a generic query for which a large number of results exists; and (3) the inexpressible query problem, in which the user cannot correctly express the query conditions to return the answer she is looking for. For each of these problems we proposed efficient solutions that focus on specific aspects of the process. We remark the main findings in the next sections and highlight the contributions and future extensions that appear as interesting research challenges.

6.1 Key Contributions

Database usability with an attention to novice users is an important aspect that needs better consideration. This dissertation makes three significant contributions in this area by proposing interactive approaches for the empty answer problem [MMBR+13, MMBR+15, MMBR+14], methods for coping with the information overload problem in graph databases [MBG15], and, finally, the exploration of exemplar queries as a solution for users struggling with inexpressible queries [MLVP14a, MLVP15, MLVP14b, LMP+14]. For each problem we provide algorithmic solutions, thorough experimental assessment, and open-source code[1]. We also built two system prototypes [MMBR+14, MLVP14b]. The details of each contribution are summarized in the following.

6.1.1 Empty Answer

We proposed a novel and principled interactive approach for queries that return no answers by suggesting relaxations to achieve a variety of optimization goals. The proposed approach follows a broad, optimization-based, probabilistic framework which takes into consideration user preferences. This is different from prior query reformulation approaches that are largely non-interactive, and/or do not support such a broad range of optimization goals. The holistic framework is adapted to return top-k reformulations at each step and to include a cardinality constraint. Moreover, any database can be translated into boolean and the same techniques can be applied.

We developed optimal and approximate solutions to the problem, demonstrating how our framework can be instantiated using different optimization goals. The solution materializes a tree together with the possible relaxations and the yes/no answers from the user, and computes a cost for each choice. In this way, at any point in the tree, the best solution that has the optimal cost can be proposed to the user.

The framework forms the basis of our IQR system prototype [MMBR+14]. We have experimentally evaluated the efficiency and effectiveness of our approach on real and synthetic datasets. We showed that our approach outperforms the previous non-interactive approaches in terms of user experience and provides timely answers on real-size datasets.

[1] Available upon request

6.1.2 Information overload

We studied the problem of query reformulation in graph databases. Graph databases are set of labeled graphs that conveniently model several data, such as chemical compounds, workflows, and ego-networks, over which a novice user is hardly able to search. Given a graph database and a query graph, the goal is to produce a set of *query reformulations*, i.e., queries that are more specific than the original query and, as such, capture a subset of the results. The reformulations represent a good abstract summary of the results as well as a useful guidance for refining the search.

We formalized the problem of finding a set of k reformulations of the input query that maximize a linear combination of coverage and diversity. We characterized its hardness and showed that it allows a greedy approximate algorithm with quality guarantees. We also devised a principled strategy to efficiently solve the most critical step of the greedy algorithm, i.e., finding the reformulation maximizing the marginal potential gain.

We experimentally validated our approaches on both real-world and synthetic databases attesting that our method runs in real-time, scales well on large databases, and provides high-quality results.

6.1.3 Inexpressible queries

Inexpressible queries are one of the most subtle obstacles for novice users who need easy ways to retrieve the data. For this issue, we introduced and defined exemplar queries, a novel query paradigm, and described how it can naturally be applied to graph data models. The exemplar queries paradigm treats the user input as an example of the intended results. This paradigm is based on the notion of similarity: an object belongs to the query answers if it is similar enough to the given user example. We formalized the similarity in two different ways, namely subgraph isomorphism and strong simulation. Subgraph isomorphism finds exact matches between the query and the graph, while strong simulation admits matches that preserve the same edge-label order but having different shapes with respect to the query. The two similarities convey issues and advantages and are used in different scenarios: isomorphism requires that the user knows exactly the example, while simulation introduces some degree of freedom.

We proposed an exact solution based on an effective and theoretically sound pruning technique, alongside an efficient approximation algorithm based on personalized PageRank. We also proved that strong-simulation algorithm can be adapted to our case exploiting a structure called expanded graph.

We evaluated our approach in terms of efficiency and effectiveness using Freebase, one of the biggest knowledge graphs available. This, as far as our knowledge goes, is the first time that Freebase is used in its entirety in the literature. We coupled our results with a user study, confirming the efficiency and usefulness of the proposed system.

6.2 Immediate Extensions

The problems presented in this dissertation have several extensions we wish to study in more detail. Possible extensions include enriching the methods to work on different data models. While this is a matter of representation and many methods to translate one format to another already exist, we would like to concentrate on fundamental questions related to database usability for novice users. In what follows, we discuss the most promising of these extensions.

6.2.1 Range and disjunctive queries

The empty answer framework deals with conjunctive queries, namely queries in which the conditions must be all satisfied at the same time. Conjunctive queries do not represent the entirety of queries that a relational database can ingest. More specifically, disjunctive queries and range conditions (i.e., conditions with $>, <, \leq, \geq$) are commonly used to specify more flexible constraints and retrieve more answers. Dealing with disjunctive queries requires a rigorous design, since the relaxation tree we propose is based on the assumption that removing one condition would result in a more generic query. This does not hold for disjunctive queries. The relaxation tree can be adapted to accept range and disjunctive queries by employing skyline queries and queries with the minimal shift from the user query [CJ09, MK09]. Therefore, instead of proposing a single relaxation, many conditions can be relaxed in a single step.

6.2.2 Query reformulation in graphs

Another interesting direction is the exploration of interactive approaches for the empty answer problem in graphs. We argue in Chapter 3 that the adaptation of existing approaches for query reformulation in graphs is not trivial. A query in a graph leads to an empty answer if it does not match any structure of the database. A relaxed query is therefore a subgraph of the user query. Finding such relaxations requires checking for isomorphic structures and, as such, is an **NP**-hard problem. To the best of our knowledge, the empty answer problem in graphs has not been studied yet. Indeed, existing solutions do not directly tackle this problem, focusing on optimizing the query answering process instead of progressively proposing more specific queries [YCHY08, YM13].

The corresponding problem of query reformulation has been studied, in this dissertation, for graph databases. The extension of this work to big graphs, such as knowledge graphs is problematic, because big graphs, conversely to graph databases, do not limit the size of the reformulations (or the limit is the entire graph). The current query reformulation solution can be applied to big graphs only considering neighborhood graphs of each node, i.e., graphs with nodes and edges at a fixed distance from a node. The latter solution is far from being perfect, since interesting patterns may not be found within a fixed distance from a node. A more promising solution comprises correlation based methods [KCN07], in which a set of interesting reformulations are chosen from those that are least correlated one another.

6.2.3 User preferences

While the empty answer framework natively makes use of user preferences, this information is not present in graph query reformulation and exemplar queries. These sources have been successfully employed in many fields [BBCV11, MPV13], for they convey valuable information about users and their habits. The exploitation of this additional information is the basis of personalized systems and optimized ranking models. Integrating such valuable sources is a challenging task, since the solutions may not scale with the size of the database and the objective functions have to adapt accordingly. The problem is then finding solutions that satisfy the user in terms of interests and preferences and

this is not an easy task. In the design of usable systems preferences should be taken into consideration.

6.2.4 Multiple exemplar queries

Multiple queries optimization has been studied extensively for performance issues in relational databases [Sel88, SG90, CM86]. However, the exemplar queries context is completely different: in the earlier works, the results for a single, or multiple queries do not change, while in exemplar queries the results become more specific as the number of example queries increases. Moreover, the straightforward solution that computes the intersection between the result sets is not directly applicable when there is no common edge labels in the input samples. A recent study computes the relevant neighborhood of the samples in order to find intersections among them [JKL$^+$14]. However, this does not provide a clear semantics for multiple exemplar queries, thus the problem needs to be investigated further.

6.3 Open Problems

This dissertation opens important research questions for the database community. Exemplar queries, interactivity, and the structured queries study, represent the tip of the database usability iceberg. Since there is a significant amount of work that remains to be done, in this section we highlight the main lines of research departing from this dissertation.

6.3.1 Probabilistic databases

The methods proposed in this dissertation do not deal with probabilistic databases. Probabilistic databases are flexible data models consisting of uncertain facts. This means that any fact in the database is associated with an existence probability. This data model comes into play when facts are extracted from sources that introduce errors, such as sensors, and data mining processes. However, flexibility comes at a price: while in deterministic databases the database has one possible instance, a probabilistic database has an exponential number of interpretations (or worlds) since any edge may

or may not exist with some probability. The proposed techniques are designed for deterministic databases, but fail to achieve the same goals with probabilistic databases. Query answering in probabilistic databases has to be completely redesigned [DS07], not only for time performance, but also because the semantics of the query and the matching function change as well.

The exploration of methods for novice users in probabilistic databases is a fresh, yet intriguing challenge. Exemplar queries, for instance, could be studied in uncertain graphs that naturally model protein interaction networks or knowledge graphs extracted from documents corpora. Query reformulation as well has to be rebuilt from scratch to work on probabilistic databases. Uncertain graphs have recently attracted the attention of the database community [BGKV14, PBGK10], but the usability of such systems is limited to few practitioners.

6.3.2 Exemplar Query Search

In Chapter 5 we presented our initial work about exemplar queries, where we show the importance of exemplar queries in the search task. Exemplar queries should become part of a research work that guides the user through all the steps, from the formulation of the query to the presentation of the results. Given a keyword query an exemplar query-enabled system should first identify if the user is asking an exemplar query. This necessitates a study of machine learning techniques to predict the user need. Second, the query is evaluated over the database using traditional keyword answering techniques, such as [PHIW12, KRS$^+$09]. In this phase multiple possible interpretations can be found. This ambiguity needs to be solved in some way, finding the best answers even when the query presents uncertainty. Third, the answers of the exemplar query are retrieved using our approach. However, multiple data sources should be employed at the same time, in order to retrieve more answers and to acquire transversal knowledge. While we considered only a single knowledge graph, searching a number of data sources at the same time introduces another level of complexity. Answering queries on multiple data sources is an interesting direction to study and deals with the traditional data integration problem [BLN86]. Finally, the results are cleaned, ranked and presented to the user. The overall process includes a number of research challenges that need to be investigated further.

Bibliography

[ABCG10] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, and Aristides Gionis. An optimization framework for query recommendation. In *WSDM*, pages 161–170, 2010.

[Abi97] Serge Abiteboul. *Querying semi-structured data.* Springer, 1997.

[Abr03] Robin Abraham. Foxq-xquery by forms. In *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*, pages 289–290. IEEE, 2003.

[ACDG03] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[ADGK09] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, and Nick Koudas. Anytime measures for top-k algorithms on exact and fuzzy data sets. *The VLDB Journal*, 18(2):407–427, 2009.

[AGHI09] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 5–14. ACM, 2009.

[BBCV11] Paolo Boldi, Francesco Bonchi, Carlos Castillo, and Sebastiano Vigna. Query reformulation mining: models, patterns, and applications. *Information retrieval*, 14(3):257–289, 2011.

[BCC05] Daniele Braga, Alessandro Campi, and Stefano Ceri. Xqbe (xq uery b y e xample): A visual interface to the standard xml query language. *ACM Transactions on Database Systems (TODS)*, 30(2):398–443, 2005.

[BDBV01] Stefano Berretti, Alberto Del Bimbo, and Enrico Vicario. Efficient matching and indexing of graph models in content-based retrieval. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(10):1089–1105, 2001.

[BDFMWB13] Ilaria Bordino, Gianmarco De Francisci Morales, Ingmar Weber, and Francesco Bonchi. From machu_picchu to rafting the urubamba river: anticipating information needs via the entity-query graph. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 275–284. ACM, 2013.

[BDG+11] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis Velegrakis. Keyword search over relational databases: a metadata approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 565–576. ACM, 2011.

[BEP+08] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1247–1250, New York, NY, USA, 2008. ACM.

[BFH07] Yvonne M Bishop, Stephen E Fienberg, and Paul W Holland. *Discrete multivariate analysis: theory and practice*. Springer Science & Business Media, 2007.

[BFJ+01] George Buchanan, Sarah Farrant, Matt Jones, Harold Thimbleby, Gary Marsden, and Michael Pazzani. Improving mobile internet usability. In *Proceedings of the 10th international conference on World Wide Web*, pages 673–680. ACM, 2001.

[BGKV14] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the*

20th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1316–1325. ACM, 2014.

[BGS14] Sonia Bergamaschi, Francesco Guerra, and Giovanni Simonini. Keyword search over relational databases: Issues, approaches and open challenges. In *Bridging Between Information Retrieval and Databases*, pages 54–73. Springer, 2014.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986.

[BLY12] Allan Borodin, Hyun Chul Lee, and Yuli Ye. Max-sum diversification, monotone submodular functions and dynamic updates. In *PODS*, pages 155–166, 2012.

[BRWD+08] S. Basu Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM*, pages 13–22, 2008.

[BYBC06] Ricardo Baeza-Yates, Paolo Boldi, and Carlos Castillo. Generalizing pagerank: Damping functions for link-based ranking algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 308–315. ACM, 2006.

[BYHM04] Ricardo A. Baeza-Yates, Carlos A. Hurtado, and Marcelo Mendoza. Query recommendation using query logs in search engines. In *EDBT Workshops*, pages 588–596, 2004.

[BYRN11] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley New York, 2011.

[Car99] Patricia Carlson. Information technology and organizational change. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 26–35. ACM, 1999.

[CDHW04] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic ranking of database query results. In *VLDB*, pages 888–899, 2004.

[CDHW06] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, 2006.

[Cha90] Surajit Chaudhuri. Generalization and a framework for query modification. In *ICDE*, pages 138–145, 1990.

[CJ09] Adriane Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[CKFY11] James Cheng, Yiping Ke, Ada Wai-Chee Fu, and Jeffrey Xu Yu. Fast graph query processing with a low-cost index. *The VLDB Journal*, 20(4):521–539, 2011.

[CKN09] James Cheng, Yiping Ke, and Wilfred Ng. Efficient query processing on graph databases. *ACM Transactions on Database Systems (TODS)*, 34(1):2, 2009.

[CKNL07] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 857–872. ACM, 2007.

[CM86] Upen S Chakravarthy and Jack Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 384–391, 1986.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Symposium on Theory of Computing*, 1971.

[CSA07] Carlos J Costa, José Silva, and Manuela Aparício. Evaluating web usability using small display devices. In *Proceedings of the 25th annual*

ACM international conference on Design of communication, pages 263–268. ACM, 2007.

[Dat83] C. J. Date. Database usability. *SIGMOD Rec.*, 13(4):1–1, May 1983.

[DP12] Marina Drosou and Evaggelia Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *Proceedings of the VLDB Endowment*, 6(1):13–24, 2012.

[DS07] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[FKM00] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. *Computer Networks*, 33(1):119–135, 2000.

[FLM+10] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment*, 3(1-2):1161–1172, 2010.

[Fri14] Thomas N Friemel. The digital divide has grown old: Determinants of a digital divide among seniors. *new media & society*, page 1461444814538648, 2014.

[FVS+09] Miquel Ferrer, Ernest Valveny, Francesc Serratosa, Itziar Bardají, and Horst Bunke. Graph-based k-means clustering: A comparison of the set median versus the generalized median graph. In *Computer Analysis of Images and Patterns*, pages 342–350. Springer, 2009.

[GFMPdlF11] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. In *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*, 2011.

[Goo14] Google. Freebase data dumps. `https://developers.google.com/freebase/data`, 2014.

[GS91] S. Gauch and J.B. Smith. Search improvement via automatic query reformulation. *TOIS*, 9(3):249–280, 1991.

[GS93] Susan Gauch and John B. Smith. An expert system for automatic query reformulation. *JASIS*, 44(3):124–136, 1993.

[GSBS03] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27. ACM, 2003.

[GXTL10] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[H+97] Dorit S Hochbaum et al. *Approximation algorithms for NP-hard problems*, volume 20. PWS publishing company Boston, 1997.

[HCY+12] Xin Huang, Hong Cheng, Jiong Yang, Jeffery Xu Yu, Hongliang Fei, and Jun Huan. Semi-supervised clustering of graph objects: a subgraph mining approach. In *Database Systems for Advanced Applications*, pages 197–212. Springer, 2012.

[HE09] Jeff Huang and Efthimis N. Efthimiadis. Analyzing and evaluating query reformulation strategies in web search logs. In *CIKM*, pages 77–86, 2009.

[HHI10] Vagelis Hristidis, Yuheng Hu, and Panagiotis G. Ipeirotis. Ranked queries over sources with boolean query interfaces without ranking support. In *ICDE*, pages 872–875, 2010.

[HWPY04] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. Spin: mining maximal frequent subgraphs from graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586. ACM, 2004.

[HWYY07] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: Ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 305–316, New York, NY, USA, 2007. ACM.

[IWM00] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data.

In *Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer, 2000.

[Jan07] Dietmar Jannach. Techniques for fast query relaxation in content-based recommender systems. *KI'06: Advances in AI*, pages 49–63, 2007.

[JCE+07] HV Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24. ACM, 2007.

[JKL+13] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. Querying knowledge graphs by example entity tuples. *arXiv preprint arXiv:1311.2100*, 2013.

[JKL+14] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. Towards a query-by-example system for knowledge graphs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[JL06] D. Jannach and J. Liegl. Conflict-directed relaxation of constraints in content-based recommender systems. *Advances in Applied AI*, pages 819–829, 2006.

[Jun04] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, volume 4, pages 167–172, 2004.

[JW03] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, pages 271–279. ACM, 2003.

[KA11] Mehdi Kargar and Aijun An. Keyword search in graphs: Finding r-cliques. *Proceedings of the VLDB Endowment*, 4(10):681–692, 2011.

[KCN07] Yiping Ke, James Cheng, and Wilfred Ng. Correlation search in graph databases. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 390–399. ACM, 2007.

[KEW09] Gjergji Kasneci, Shady Elbassuoni, and Gerhard Weikum. Ming: Mining informative entity relationship subgraphs. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1653–1656, New York, NY, USA, 2009. ACM.

[KK01] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 313–320. IEEE, 2001.

[KLTV06] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[KLY+11] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. Neighborhood based fast graph search in large networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 901–912. ACM, 2011.

[KRS+09] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M Suchanek, and Gerhard Weikum. Star: Steiner-tree approximation in relationship graphs. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 868–879. IEEE, 2009.

[Kun90] Hideko S Kunii. Graph data model. In *Graph Data Model*, pages 7–20. Springer, 1990.

[KWAY13] Arijit Khan, Yinghui Wu, Charu C Aggarwal, and Xifeng Yan. Nema: Fast graph search with label similarity. *Proceedings of the VLDB Endowment*, 6(3):181–192, 2013.

[LC10] Ni Lao and William W Cohen. Fast query execution for retrieval models based on path-constrained random walks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 881–888. ACM, 2010.

[LIJ+15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[LMP+14] Matteo Lissandrini, Davide Mottin, Dimitra Papadimitriou, Themis Palpanas, and Yannis Velegrakis. Unleashing the power of information graphs. *SIGMOD Record*, 43(4), 2014.

[LXCB12] Wenqing Lin, Xiaokui Xiao, James Cheng, and Sourav S Bhowmick. Efficient algorithms for generalized subgraph query processing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 325–334. ACM, 2012.

[LYR+10] C. Li, N. Yan, S. Basu Roy, L. Lisham, and G. Das. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *WWW*, pages 651–660, 2010.

[MBG15] Davide Mottin, Francesco Bonchi, and Francesco Gullo. Graph query reformulation with diversity. In *Proceedings of the 21st ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2015.

[MCF+14] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 39(1):4, 2014.

[MD79] R Garey Michael and S Johnson David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MK09] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873. ACM, 2009.

[MLVP14a] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5), 2014.

[MLVP14b] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Searching with xq: The exemplar query search engine. In *SIGMOD*, pages 901–904, New York, NY, USA, 2014. ACM.

[MLVP15] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: a new way of searching. *Submitted for publication*, 2015.

[MMBR+13] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A probabilistic optimization framework for the empty-answer problem. *PVLDB*, 6(14), 2013.

[MMBR+14] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. Iqr: An interactive query relaxation system for the empty-answer problem. In *SIGMOD*, pages 1095–1098, New York, NY, USA, 2014. ACM.

[MMBR+15] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A holistic and principled approach for the empty-answer problem. *Submitted for publication*, 2015.

[Mot84] Amihai Motro. Query generalization: A method for interpreting null answers. In *Expert Database Workshop*, pages 597–616. Citeseer, 1984.

[MPV13] Davide Mottin, Themis Palpanas, and Yannis Velegrakis. Entity ranking using click-log information. *Intelligent Data Analysis*, 17(5):837–856, 2013.

[MS00] Shinichi Morishita and Jun Sese. Transversing itemset lattices with statistical metric pruning. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 226–236. ACM, 2000.

[MWU+07] Thorsten Meinl, Marc Wörlein, Olga Urzova, Ingrid Fischer, and Michael Philippsen. The parmol package for frequent subgraph mining. *Electronic Communications of the EASST*, 1, 2007.

[NK04] Siegfried Nijssen and Joost N Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652. ACM, 2004.

[ora] Oracle developer tools. `http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html`. Accessed: 2015-02-22.

[ORPF13] Alper Okcan, Mirek Riedewald, Biswanath Panda, and Daniel Fink. Scolopax: exploratory analysis of scientific data. *Proceedings of the VLDB Endowment*, 6(12):1298–1301, 2013.

[Par81] David Park. *Concurrency and automata on infinite sequences*. Springer, 1981.

[PBGK10] Michalis Potamias, Francesco Bonchi, Aristides Gionis, and George Kollios. K-nearest neighbors in uncertain graphs. *Proceedings of the VLDB Endowment*, 3(1-2):997–1008, 2010.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[PHIW12] Jeffrey Pound, Alexander K Hudek, Ihab F Ilyas, and Grant Weddell. Interpreting keyword queries over web knowledge bases. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 305–314. ACM, 2012.

[PKM05] Themis Palpanas, Nick Koudas, and Alberto Mendelzon. Using datacube aggregates for approximate querying and deviation detection. *Knowledge and Data Engineering, IEEE Transactions on*, 17(11):1465–1477, 2005.

[PPV02] Yannis Papakonstantinou, Michalis Petropoulos, and Vasilis Vassalos. Qursed: querying and reporting semistructured data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 192–203. ACM, 2002.

[RHS14] Sayan Ranu, Minh Hoang, and Ambuj Singh. Answering top-k representative queries on graph databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1163–1174. ACM, 2014.

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[Sel88] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[SG90] Timos Sellis and Subrata Ghosh. On the multiple-query optimization problem. *Knowledge and Data Engineering, IEEE Transactions on*, 2(2):262–266, 1990.

[Sha01] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[SLZ+10] Haichuan Shang, Xuemin Lin, Ying Zhang, Jeffrey Xu Yu, and Wei Wang. Connected substructure similarity search. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 903–914. ACM, 2010.

[SM95] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., 1995.

[SMG09] Bingjun Sun, Prasenjit Mitra, and C Lee Giles. Independent informative subgraph mining for graph information retrieval. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 563–572. ACM, 2009.

[SMO10] Rodrygo LT Santos, Craig Macdonald, and Iadh Ounis. Exploiting query reformulations for web search result diversification. In *Proceedings of the 19th international conference on World wide web*, pages 881–890. ACM, 2010.

[sql] Microsoft sql server data tools. `https://msdn.microsoft.com/en-us/data/tools.aspx`. Accessed: 2015-02-22.

[SY98] Roberta Evans Sabin and Tieng K Yap. Integrating information retrieval techniques with traditional db methods in a web-based database

browser. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 760–766. ACM, 1998.

[SZLY08] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[TC10] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[TS05] Diana Tabatabai and Bruce M Shore. How experts and novices search the web. *Library & information science research*, 27(2):222–248, 2005.

[Tuk77] John W Tukey. Exploratory data analysis. *Reading, Ma*, 231:32, 1977.

[TY09] Yufei Tao and Jeffrey Xu Yu. Finding frequent co-occurring terms in relational keyword search. In *EDBT*, pages 839–850, 2009.

[Ull76] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[Val79] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[WDT+12] Xiaoli Wang, Xiaofeng Ding, A Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 210–221. IEEE, 2012.

[WNZ02] Ji-Rong Wen, Jian-Yun Nie, and HongJiang Zhang. Query clustering using user logs. *ACM Trans. Inf. Syst.*, 20(1):59–81, 2002.

[WZ08] Xuanhui Wang and ChengXiang Zhai. Mining term association patterns from search logs for effective query reformulation. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 479–488. ACM, 2008.

[YCHH12] Junjie Yao, Bin Cui, Liansheng Hua, and Yuxin Huang. Keyword query reformulation on structured data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 953–964. IEEE, 2012.

[YCHY08] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S Yu. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 433–444. ACM, 2008.

[YH02] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[YM13] Dayu Yuan and Prasenjit Mitra. Lindex: a lattice-based index for graph databases. *The VLDB Journal*, 22(2):229–252, 2013.

[YQC09] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in databases. *Synthesis Lectures on Data Management*, 1(1):1–155, 2009.

[YWSY14] Shengqi Yang, Yinghui Wu, Huan Sun, and Xifeng Yan. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment*, 7(7), 2014.

[YYH04] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346. ACM, 2004.

[YYH05] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing based on discriminative frequent structure analysis. *ACM Transactions on Database Systems (TODS)*, 30(4):960–993, 2005.

[Zic13] Kathryn Zickuhr. Who's not online and why. *Pew Internet & American Life Project*, page 40, 2013.

[Zlo75] Moshé M. Zloof. Query by example. In *AFIPS NCC*, pages 431–438, 1975.

[ZTR13] Lei Zhang, Thanh Tran, and Achim Rettinger. Probabilistic query rewriting for efficient and effective keyword search on graph data. *Proceedings of the VLDB Endowment*, 6(14):1642–1653, 2013.