Querying Graphs with Preferences

Valeria Fionda
KRDB, Free University of Bozen-Bolzano
Piazza Domenicani 3
Bolzano, Italy
fionda@inf.unibz.it

Giuseppe Pirrò
KRDB, Free University of Bozen-Bolzano
Piazza Domenicani 3
Bolzano, Italy
pirro@inf.unibz.it

ABSTRACT

This paper presents GuLP a graph query language that enables to declaratively express preferences. Preferences enable to order the answers to a query and can be stated in terms of nodes/edge attributes and complex paths. We present the formal syntax and semantics of GuLP and a polynomial time algorithm for evaluating GuLP expressions. We describe an implementation of GuLP in the GuLP-IT system, which is available for download. We evaluate the GuLP-IT system on real-world and synthetic data.

Categories and Subject Descriptors

H.2.3 [**DATABASE MANAGEMENT**]: Languages; H.2.4 [**DATABASE MANAGEMENT**]: Systems

Keywords

Graph Query Languages; Preferences

1. INTRODUCTION

"I like Rock music better than Pop". This is a natural and intuitive way of expressing preferences. The underlying semantics of the "better than" tells us that not only we prefer Rock music to Pop but also that sometimes we may listen to Pop music. Generally speaking, this kind of preferences are said to be qualitative as opposed to quantitative. The difference consists in the fact that in the first case we are specifying a particular musical gender while in the second case the specification has to be accompanied by a quantitative estimation (e.g., *Rock* 0.7, *Pop* 0.3). Qualitative preferences are strictly more general than the quantitative ones; this is because it is always possible to "convert" scores into preference relations but not vice versa [6]. Expressing preferences is of paramount importance in many fields and especially when dealing with large volumes of data (e.g., [10, 16, 20, 19]). Instead of providing a huge and unordered result set, it is more useful to provide a way to distinguish between more and less *preferred* answers. This fostered research in database theory about how to incorporate preferences in query languages (e.g., Kiessling [14] Chomicki [7]).

Although relational databases maintain an important role in data-intensive tasks, we are assisting to a deluge of data represented as graphs. Graphs are a powerful data structure enabling to represent in a natural way objects along with their relations. Social networks, biological networks and projects such as the Google Knowledge Graph and Linked Data on the Web [5], give just a glimpse of the magnitude and spread of graph-like data. Hence, as in the relational world, the problem of expressing preferences is becoming more and more relevant. Despite the variety of graph query languages and tools today available (see Angles and Gutierrez [2] and Wood [22] for comprehensive surveys), little has been done in terms of specific features enabling to express preferences. Some approaches have been proposed that enable expressing quantitative preferences only over graph edges (e.g., [13]). Other initiatives consider qualitative preferences and distinguish levels of preference by means of partial orders (e.g., [12]). Besides, the simple data model (i.e., graphs with no attributes) and query language (i.e., standard regular expressions) considered, these approaches do not enable to express complex preferences. Last but not least, we are not aware of any implementation of graph query languages with preferences.

In this paper we focus on the following research questions: why are preferences useful in a graph query language? How to declaratively express preferences? At which level of granularity the user should be able to express preferences (e.g., only edges, entire paths, attributes)? Is it always useful to embed preferences in the language or in some cases it is better to execute a set of queries each expressing a level of preference?

Having preferences in a graph query language enables to drive the evaluation to consider some paths before than others thus introducing an ordering on the way results are collected and presented. This functionality is useful when one is interested in certain kinds of results more than in others and is crucial when dealing with large graphs. Motivated by this consideration, we face the problem of embedding preferences in a declarative language on the form of edge types, node and edge attributes and entire paths. To enable such a rich variety of preferences we introduce the $\succ$ operator in graph navigational languages. As underlying data model we consider *attributed graphs* that generalize other models by enabling attributes on both nodes and edges.

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2505515.2505758.

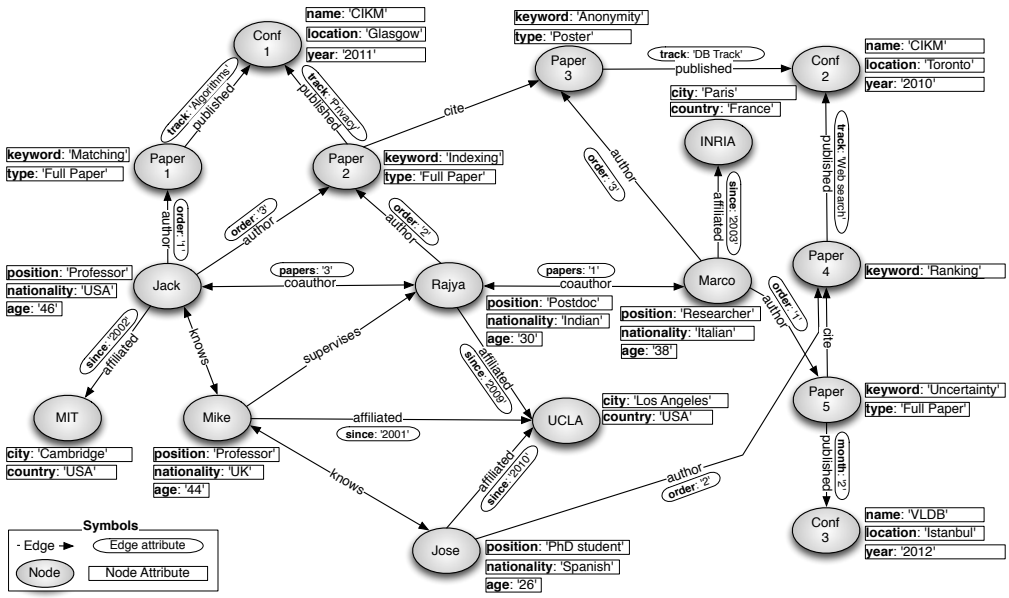


Figure 1: An example of attributed graph. Both nodes and edges have attributes.

An example of attributed graph is depicted in Fig. 1, where nodes represent different kinds of objects such as persons or publications, node attributes allow to specify their properties, edges represent the relations between objects and edge attributes enable to further refine these relations. Given a graph (e.g., Fig. 1), one would like to pose queries like *"find all the co-authors of Jack"* or *"people known by Jack or by someone Jack knows"*. These are typical *reachability* queries asking for the existence of paths from a source node (i.e., *Jack*) toward other nodes. This kind of requests can be fulfilled by one of the existing graph query languages [2, 22]. However, current languages do not enable to pose requests such as "*find papers authored by someone related to Jack that published at CIKM but prefer Jack's co-authors to persons merely known by Jack*".

The aim of this paper is to present GuLP(Graph Query Language with Preferences) that enables to fulfill such kinds of requests. In the previous examples, with GuLP one would obtain first the set {*Paper2, Paper3, Paper5*} and then {*Paper4*}. The flexibility of GuLP enables to declaratively specify preferences in terms of path types (e.g., prefer *co-author* edges to *knows* edges), node and edge attributes (e.g., prefer *professors* to *PhD students* or co-authors with 3 papers). The benefit of a system implementing GuLP is twofold. First, it enables to present results according to user preferences declaratively specified. Second, the system could implement an any-time behavior. This means that the user can start receiving the most preferred results and stop the system at any point of the computation.

1.1 GuLP by example

To give a hint on the features of GuLP we discuss some examples considering the graph in Fig. 1. The formal syntax and semantics of GuLP are introduced in Section 3.

Example 1.1. ***(Preference based on path labels)*** *Find papers published by persons linked to Jack by only* `co-author` *or only* `knows` *paths giving preference to co-author paths.*

The request in the previous example differs from a classical reachability query in one important respect: it defines a preference, which enables to distinguish nodes in the results. In GuLP it can be expressed as follows:

$$\succ(\texttt{co-author}^*, \texttt{knows}^*)/\texttt{author}$$

In the expression, the operator $\succ(pref_1, pref_2, ...pref_k)$ is used to declaratively specify preferences. Generally speaking, each $pref_i$ can be an edge type, a path or a test over node or edge attributes. In this example paths defined by `co-author`* are preferred to those defined by `knows`*. Hence, two levels of preference are defined; the first is associated to the expression `co-author`*`/author` while the second one is associated to `knows`*`/author`.

The evaluation of a GuLP expression returns an ordered list L of sets of nodes. Each set in the list maintains nodes that have been *reached* at a certain *level* of preference. Moreover, each node in the results only appears in the highest preference set. The result of the evaluation is L = [*Pref1*:{*Paper1*,*Paper2*,*Paper3*, *Paper5*}, *Pref2*:{*Paper4*}]. Note that the node labeled as *Paper1* can be reached in both levels of preference (i.e., via both types of paths) but it is considered only at the highest level.

Example 1.2. ***(Preference based on edge and node attributes)*** *Find conferences to which the co-authors of Rajya have published and present these conferences first by the co-authorship degree and then by the type of publication.*

This request defines preferences through a combination of edge attributes, to define the co-authorship degree, and node attributes to prefer conferences on the basis of the kind of paper published. If *Full Papers* are preferred to *Posters*, the specification in GuLP is:

$$\succ(\texttt{coauthor}\{\texttt{papers}{>}1\}, \texttt{coauthor}\{\texttt{papers}{=}1\})/$$
$$\succ(\texttt{author}.\{\texttt{type}{=}\text{'Full'}\}, \texttt{author}.\{\texttt{type}{=}\text{'Poster'}\})/\texttt{published}$$

The evaluation of the first part of the expression starting from the node *Rajya* distinguishes nodes reachable considering the preference (defined via $\succ$) on the attribute `papers` of `coauthor` edges. The most preferred nodes are those reachable via `coauthor` edges having the attribute `papers`>1; then, nodes where the same edge attribute is equal to 1 are considered. The evaluation continues with the second part of the expression where the node attribute `type`, reached via `author` edges, is used for expressing preferences. The last part considers conference nodes reachable through an edge `published`. Fig. 2 provides a high level overview of the evaluation of this expression.

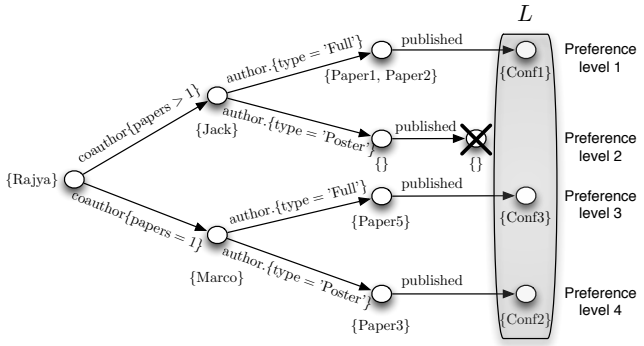


Figure 2: An overview of GuLP.

By looking at Fig. 2 one would be tempted to rewrite a GuLP expression with preferences into set of expressions without preferences and then evaluate them separately. As we will show in Section 5 (Experiment 2) this approach has an exponential running time while GuLP expressions can be evaluated in polynomial time.

Example 1.3. ***(Preference based on edge and node attributes plus nesting)*** *Find papers, published in 2012, by someone who published at least one paper at CIKM and is linked to Jack by a path of co-author edges or knows edges. Give preference to the first type of paths.*

This request involves preferences and nesting and can can be specified in GuLP as follows:

$$\succ(\texttt{co-author}^*, \texttt{knows}^*)/[\texttt{author/published.}\{\texttt{name='CIKM'}\}]/\texttt{author}[\texttt{published.}\{\texttt{year='2012'}\}]$$

The result of the evaluation of the subexpression `co-author`* is *Pref1*={*Jack*, *Rajya*, *Marco*}. As for the subexpression `knows`* the result is *Pref2*={*Mike*, *Jose*}. Results at each of the two levels of preference are filtered via the nested expression [`author/published.{name='CIKM'}`], which keeps only authors of CIKM papers. After the evaluation of the nested expression we have at the first level of preference *Pref1*={*Jack*, *Rajya*, *Marco*}, since *Jack* authored *Paper1* and *Paper2*, *Rajya Paper2* and *Marco Paper3*, all published at CIKM. As for the second level of preference, GuLP discards *Mike* since he did not publish any paper at CIKM, and keeps only *Jose* who authored *Paper4*, thus obtaining *Pref2*={*Jose*}. The evaluation of the subexpression `author` considers all the papers of such persons thus giving *Pref1*={*Paper1*,*Paper2*, *Paper3*, *Paper5*}, *Pref2*={*Paper4*}. The last nested expression discards all papers that have not been published in 2012, giving: *Pref1*={*Paper5*}, *Pref2*={}.

1.2 Related work

The problem of expressing preferences has been addressed in [8, 10]. Foundational preference frameworks for databases have been introduced in [7, 14, 16, 20]. The aim is to enable to express preferences over tuples. Although these approaches were source of inspiration for our preference framework, our departure point is different: we focus on defining a declarative language for querying graphs with preferences while these approaches focus on how to compute preferred tuples given preferences on attributes (not entire complex paths). Limited form of preferences have been considered for XPath (e.g., [1, 15]). As for graphs, despite the variety of graph query languages and tools today available [2, 22], little has been done in terms of specific features enabling to express preferences. Let's look at the limitations of the state of the art by considering the graph in Fig. 1. Almost all graph languages (e.g., [17]) enable to fulfill requests such as *"find the conferences where co-authors of Rajya have published"* where all nodes reachable are considered equally preferable. However, when it comes to requests such as *"prefer conferences held in 2012 and then those held in 2010"* or *"find people connected to Mike but prefer first professors he knows and then people he supervises"* current approaches fail short. Vidal et al. [21] define a method based on a heuristic to chose paths from a graph. This method does not allow to declaratively express preferences. Grahne et al. [13] presented an approach, which enables the user to assign natural numbers (i.e., *quantitative* preferences) to graph edges and compute the answer in terms of weighted paths. As for the previous examples, this approach does not enable to express preferences in terms of node and edge attributes. Moreover, no preferences over whole paths can be expressed. Flesca and Greco [12] introduced partially ordered regular languages with which it is not always possible to obtain a unique ordering of the results. Besides, the data model considered and the language are less expressive than those considered in this paper. For instance with this approach it is not possible to express requests such as "*Find papers, published in 2012, by someone who published at least one paper at CIKM and is linked to Jack by a path of coauthor edges or knows edges. Give preference to the first type of paths*".

This analysis motivates the need for a comprehensive solution to the problem of declaratively expressing preferences in graph languages. We identified the following desiderata in designing GuLP: *i) declarative specification*: the user should declaratively express preferences so that nodes in the answers to a query can be presented in order of preference; *ii) rich preference model*: preferences should be expressed at various levels ranging from simple edge types to node and edge attributes and entire paths; *iii) efficiency*: GuLP expressions should be evaluated with a low polynomial combined complexity.

1.3 Contributions and roadmap

The following are the main contributions of this paper:

1. *We introduce* GuLP *(Graph Query Language with Preferences)* to declaratively express preferences over graph edges, node and edge attributes and entire paths.
2. *We provide* a simple and concise syntax and a clear semantics for GuLP.
3. *We devise an algorithm* to evaluate GuLP expressions and study its complexity.

4. *We implement* GuLP in the GuLP-IT system available at `http://graphgulp.wordpress.com`.
5. *We evaluate* GuLP-IT on both real-world and synthetic datasets.

The paper is organized as follows. Section 2 introduces the data model and provides an overview on graph query languages based on regular expressions. In Section 3 GuLP is introduced: syntax, semantics and the preference operator. Here it is also discussed the pseudo-code of the algorithm for evaluating GuLP expressions and its complexity analysis. Section 5 discusses the implementation and evaluation of GuLP on both a real-world and synthetic datasets. Finally, Section 6 concludes the paper and sketches future work.

2. PRELIMINARIES

Data Model. An attributed graph is a directed labeled multi-graph of the form $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}, \mathcal{F}_n, \mathcal{F}_e, \mathcal{A}, \mathcal{V}_n, \mathcal{V}_e \rangle$, where $\mathcal{N}$ is a set of nodes, $\mathcal{E}$ a set of edges and $\mathcal{A}$ a set of attribute types. An edge $e \in \mathcal{E}$ is a triple of the form $\langle n_{out}, \lambda_e, n_{in} \rangle$ where n_{out} and n_{in} are the outgoing and ingoing nodes and $\lambda_e \in \Sigma$ is the label associated to e from a finite alphabet Σ. The function $\mathcal{F}_n : \mathcal{N} \rightarrow 2^{\mathcal{A}}$ assigns to each node a subset of attributes in $\mathcal{A}$. Similarly it does $\mathcal{F}_e : \mathcal{E} \rightarrow 2^{\mathcal{A}}$ for edges. Function $\mathcal{V}_n$ assigns to each pair (n, a) s.t. $n \in \mathcal{N}$ and $a \in \mathcal{F}_n(n)$ the value of attribute a for node n. Similarly it does $\mathcal{V}_e$ for edges. As an example $\mathcal{F}_n$ assigns to the node *Conf1* in Fig. 1 some attributes such as `location` and `year`, and $\mathcal{V}_n$ assigns the value *Glasgow* to the pair (*Conf1*, `location`) and the value *2011* to the pair (*Conf1*, `year`). The functions $\mathcal{F}_e$ and $\mathcal{V}_e$ do the same for labeled edges e.g., the edge `published`, between *Paper2* and *Conf1*, has attribute `track` with value *Privacy*.

Languages based on Regular Expressions. In this paper we focus on the family of *reachability* queries. Essentially, a reachability query aims at selecting nodes in the graph that are reachable from a given node via paths (i.e., strings obtained by concatenating edge labels) that belong to the language of regular expressions defined over Σ. As an example, the query $\texttt{knows}^*$ starting from *Jack* in the graph in Fig. 1 enables to reach the nodes *Mike* and *Jose*. The reader can refer to [2] for further details. In particular, we consider Nested Regular Expressions (NREs) [17] as backbone for the GuLP language.

3. THE GuLP LANGUAGE

GuLP is a graph query language for attributed graphs that enables expressing preferences via the preference operator $\succ$. In this section we present the syntax and semantics of the language.

3.1 Syntax

The syntax of GuLP is reported in Table 1. The building blocks of a GuLP expression are edge labels and tests over attribute values. The symbol _ is a wildcard to indicate any edge label. Paths can be *preference* paths (`ppath`), that contain the preference operator $\succ$, or *standard* paths (`spath`) that can be nested. A preference path is an expression of the form $\succ$ $(\texttt{ppath}_1, ..., \texttt{ppath}_k)$ where the various $\texttt{ppath}_i$ are given in order of preference.

Table 1: The syntax of GuLP

`ppath`	::= `spath` | `ppath/ppath` | `(ppath)`* | `ppath[spath]` | `ppath.{test}` | $\succ$ $(\texttt{ppath}_1, ..., \texttt{ppath}_n)$
`spath`	::= `edge` | `spath/spath` | `spath|spath` | `(spath)`* | `spath[spath]` | `spath.{test}`
`edge`	::= `label` | `label{test}`
`label`	::= l | l^{-1} | _ | _$^{-1}$
`test`	::= *attr* `oper` *value* | `test` $\wedge$ `test` | `test|test`
`oper`	::= $>\|<\|=\|\neq$

3.2 Semantics of GuLP

To introduce the semantics of the language, reported in Table 2, consider the following expression $\succ$ $(\texttt{ppath}_1, \texttt{ppath}_2)$. The meaning of this expression recalls that of "$\texttt{ppath}_1$ better than $\texttt{ppath}_2$" in natural language, that is, "I prefer to see nodes reachable from a given node n via $\texttt{ppath}_1$ first and then (separately) I also like to see nodes reachable via $\texttt{ppath}_2$". This corresponds to define two *levels* of preference: one for the set of nodes reachable via $\texttt{ppath}_1$ and one for the set of nodes reachable via $\texttt{ppath}_2$ (and not $\texttt{ppath}_1$). At each of the two levels of preference there can be more than one node and the same node cannot belong to more than one level of preference. This reasoning can be generalized to expressions of the form $\succ$ $(\texttt{ppath}_1, ..., \texttt{ppath}_n)$.

To deal with tests on node/edge attributes, an auxiliary function `EvalT` is exploited, which checks that the attributes match the criteria expressed in `test`. As anticipated earlier, given a GuLP expression $e^{\succ}$, an attributed graph $\mathcal{G}$ and a seed node n, the evaluation of $e^{\succ}$ over $\mathcal{G}$ from n returns a list L of sets of nodes, reachable from n, where each set maintains nodes at a certain level of preference. To refer to the i-th set in the list we use the notation $L\langle i \rangle$. It holds that $\forall i, j \; L\langle i \rangle \cap L\langle j \rangle = \emptyset$. Hence, the evaluation of a GuLP expression returns not only the most preferred results, but also results at lower levels of preference. In the case of a *standard path* (`spath`), that is, no preferences, there will be only one set. To give a hint of how the preference operator works, consider the following expression:

$$Jack \; (\texttt{co-author} \mid \texttt{knows})/ \succ (\texttt{knows}, \texttt{supervises})$$

where *Jack* is the seed node. When evaluating this expression on the graph reported in Fig. 1, nodes reachable via an edge `co-author` or an edge `knows`, that is, $\{Rajya, Mike\}$ are equally preferable. Starting from $\{Rajya, Mike\}$, *Jose* is reached via the more preferable edge `knows` while *Rajya* via the less preferable edge `supervises`. Overall, the result consists in two sets $S_1 = \{Jose\}$ and $S_2 = \{Rajya\}$ with nodes in S_1 being more preferable than nodes in S_2.

The semantics of a GuLP expression recalls that of a total order; each pair of nodes appearing in some sets in L can be compared. We say that a node $n_q \in L\langle i \rangle$ is preferred to another node $n_r \in L\langle j \rangle$, denoted by $n_q \succ n_r$, if $i < j$. Two nodes are equally preferable, denoted by $n_q \sim n_r$, if $i = j$ (i.e., they belong to the same set). The rational behind using a total order stems from the following observation: when a user queries a search engine, his preferences (i.e., keywords) are matched against a collection of documents with relevant documents presented as a totally ordered set. We apply the same reasoning here with the difference that GuLP instead of keywords deals with regular expressions and instead of a set of documents returns a list of sets of nodes.

Table 2: The semantics of GuLP. Function `EvalT` checks that node or edge attributes satisfy the `test`.

$$[\![l]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) | \exists \langle \mathtt{n}, l, \mathtt{y}\rangle \in \mathcal{G}\}]$$

$$[\![l^{-1}]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) | \exists \langle \mathtt{y}, l, \mathtt{n}\rangle \in \mathcal{G}\}]$$

$$[\![\_]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) | \exists \langle \mathtt{n}, \_, \mathtt{y}\rangle \in \mathcal{G}\}] \text{ /similarly } \_^{-1}$$

$$[\![l\{\mathtt{test}\}]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) | \exists \mathtt{e}{=}\langle \mathtt{n}, l, \mathtt{y}\rangle \in \mathcal{G} \wedge \mathtt{EvalT}(\mathtt{test}, \mathtt{e.attributes})\}] \text{ /similarly } l^{-1}, \_ \text{ and } \_^{-1}$$

$$[\![(\mathtt{spath}_1/\mathtt{spath}_2)]\!]^{\mathtt{n}}_{\mathcal{G}} = [(\mathtt{n},\mathtt{y}) \mid \exists \mathtt{z} \; s.t. \; (\mathtt{n},\mathtt{z}) \in [\![\mathtt{spath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \wedge (\mathtt{z},\mathtt{y}) \in [\![\mathtt{spath}_2]\!]^{\mathtt{z}}_{\mathcal{G}}{<}0{>}]$$

$$[\![(\mathtt{ppath}_1/\mathtt{ppath}_2)]\!]^{\mathtt{n}}_{\mathcal{G}} = [S_1, ..., S_{q\times p}] \mid (\mathtt{n},\mathtt{y}) \in S_j \text{ if } \exists \mathtt{z} \; s.t. \; (\mathtt{n},\mathtt{z}) \in [\![\mathtt{ppath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}{<}\mathrm{i}{>} \wedge (\mathtt{z},\mathtt{y}) \in [\![\mathtt{ppath}_2]\!]^{\mathtt{z}}_{\mathcal{G}}{<}\mathrm{k}{>} \wedge$$
$$j = [(i-1)\times p + k] \wedge \forall l \leq j \; (\mathtt{n},\mathtt{y}) \notin S_l \wedge p = |\text{preference levels of } \mathtt{ppath}_2|\}$$

$$[\![(\mathtt{spath}_1|\mathtt{spath}_2)]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) | (\mathtt{n},\mathtt{y}) \in [\![\mathtt{spath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \vee (\mathtt{n},\mathtt{y}) \in [\![\mathtt{spath}_2]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>}\}]$$

$$[\![(\mathtt{spath})^*]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{n})\} \cup \bigcup_{i=1}^{\infty} [\![\mathtt{spath}_i]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \mid \mathtt{spath}_1 = \mathtt{spath} \wedge \mathtt{spath}_i = \mathtt{spath}_{i\text{-}1}/\mathtt{spath}]$$

$$[\![(\mathtt{ppath})^*]\!]^{\mathtt{n}}_{\mathcal{G}} = [S_1, ..., S_m, ...] \mid S_1 = \{(\mathtt{n},\mathtt{n})\} \cup [\![\mathtt{ppath}]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \cup \{(\mathtt{n},\mathtt{z}) \mid \exists (\mathtt{n},\mathtt{y}) \in [\![\mathtt{ppath}]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \; s.t. \; (\mathtt{y},\mathtt{z}) \in [\![\mathtt{ppath}]\!]^{\mathtt{y}}_{\mathcal{G}}{<}0{>}\} \cup \; ...,$$
$$S_m = [\![\mathtt{ppath}]\!]^{\mathtt{n}}_{\mathcal{G}}{<}\mathrm{m}{>} \cup \{(\mathtt{n},\mathtt{z}) \mid \exists (\mathtt{n},\mathtt{y}) \in [\![\mathtt{ppath}]\!]^{\mathtt{n}}_{\mathcal{G}}{<}\mathrm{last}{>} \; s.t. \; (\mathtt{y},\mathtt{z}) \in [\![\mathtt{ppath}]\!]^{\mathtt{y}}_{\mathcal{G}}{<}0{>}\} \cup \; ...$$

$$[\![\succ (\mathtt{ppath}_1, ..., \mathtt{ppath}_k)]\!]^{\mathtt{n}}_{\mathcal{G}} = L = [S_1, S_2, ..., S_m, ..., S_q] \mid \forall_{0\leq i\leq [\![\mathtt{ppath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}.size} \; S_i = [\![\mathtt{ppath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}{<}i{>} \wedge \forall_{2\leq i\leq k} \; append\&Remove(L, [\![\mathtt{ppath}_i]\!]^{\mathtt{n}}_{\mathcal{G}})$$

$$[\![\mathtt{spath}_1[\mathtt{spath}_2]]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) \mid (\mathtt{n},\mathtt{y}) \in [\![\mathtt{spath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \wedge \exists \mathtt{z} \; s.t. \; (\mathtt{y},\mathtt{z}) \in [\![\mathtt{spath}_2]\!]^{y}_{\mathcal{G}}{<}0{>}\}]$$

$$[\![\mathtt{ppath}_1[\mathtt{spath}_2]]\!]^{\mathtt{n}}_{\mathcal{G}} = [S_1, ..., S_i, ..., S_m] \mid (\mathtt{n},\mathtt{y}) \in S_i \text{ if } (\mathtt{n},\mathtt{y}) \in [\![\mathtt{ppath}_1]\!]^{\mathtt{n}}_{\mathcal{G}}{<}\mathrm{i}{>} \wedge \exists \mathtt{z} \; s.t. \; (\mathtt{y},\mathtt{z}) \in [\![\mathtt{spath}_2]\!]^{y}_{\mathcal{G}}{<}0{>}\}/\text{similarly } \mathtt{spath}_1[\mathtt{spath}_2]$$

$$[\![\mathtt{spath.\{test\}}]\!]^{\mathtt{n}}_{\mathcal{G}} = [\{(\mathtt{n},\mathtt{y}) | (\mathtt{n},\mathtt{y}) \in [\![\mathtt{spath}]\!]^{\mathtt{n}}_{\mathcal{G}}{<}0{>} \wedge \mathtt{EvalT}(\mathtt{test}, \mathtt{y.attributes})\}]$$

$$[\![\mathtt{ppath.\{test\}}]\!]^{\mathtt{n}}_{\mathcal{G}} = [S_1, ..., S_i, ..., S_m] \mid (\mathtt{n},\mathtt{y}) \in S_i \text{ if } (\mathtt{n},\mathtt{y}) \in [\![\mathtt{ppath}]\!]^{\mathtt{n}}_{\mathcal{G}}{<}\mathrm{i}{>} \wedge \mathtt{EvalT}(\mathtt{test}, \mathtt{y.attributes})\}/\text{similarly } \mathtt{spath.\{test\}}$$

If one considers a *partial* order over the results then this reasoning does not apply any more since not all nodes are comparable [12]. As it can be observed in Table 1, there is a distinction between `ppath` and `spath` since the operator $\succ$ is not allowed in nested expressions and expressions involving union. We motivate this choice in the following.

Preferences in nesting. We consider the original semantics of NREs according to which a nested expression included between [ ] corresponds to an existential test. This amounts at checking whether the result of this expression is not empty. With this reasoning, the evaluation of an expression of the form $[\succ (\mathtt{ppath}_1, ..., \mathtt{ppath}_k)]$ would check if some results exist (no matter their preference level). There is another possible semantics for handling preferences in the nesting: the evaluation of $[\succ (\mathtt{ppath}_1, ..., \mathtt{ppath}_k)]$ outputs the highest level of preference for which some results exist. This allows to order the results also on the basis of the nested expressions. In the current implementation we consider the first semantics and leave the second one as future work.

Union of preferences. Expressions dealing with the union of *preference paths* are not allowed as they violate the total order induced by $\succ$. Consider the following expression evaluated over the graph in Fig. 1:

```
(coauthor*/≻(author.{type='Full'},author.{ type='Poster'}))|
  (knows*/≻(author.{type='Full'},author.{ type='Poster'}))
```

The operator | represents the union of the results of the two sub-expressions. From the first, we obtain a list of two sets L_1=[{Paper2,Paper5}, {Paper3}]. The second one produces only one set L_2=[{Paper4}]. As it can be noted, it is not possible to perform the union while preserving the properties of the total order imposed by the $\succ$ operator. For instance, nothing can be said about the level of preference of Paper4 $\in L_2\langle 0\rangle$ w.r.t. elements in $L_1\langle 0\rangle$ and $L_1\langle 1\rangle$.

Why embedding the operator $\succ$ in the language? The motivation is threefold: *i)* GuLP is able to return results already ordered in different levels of preference when evaluating an expression. If one wanted to apply preferences after collecting the results it would be necessary to "remember" all the paths toward some result, which hinders the complexity of the language [4]; *ii)* the declarative specification of preferences enables to distinguish the *preferred* parts within the whole answer set beforehand; *iii)* an *any-time* implementation of the GuLP language gives to the user the possibility to stop the evaluation after receiving enough *preferred* results without the need to collect all the results and then apply the preferences. This is useful when dealing with large graphs or Web navigational languages [11].

4. ALGORITHMS AND COMPLEXITY

This section presents a polynomial algorithm for the evaluation of GuLP expressions. In the following we indicate by $e^{\succ}$ a GuLP expression that may contain the preference operator $\succ$ and by e an expression without preferences.

The evaluation of a GuLP expression starts by invoking the function Evaluate. It receives in input an attributed graph $\mathcal{G}$, a GuLP expression $e^{\succ}$ and a seed node n (given as I=[{n}]). The result of the evaluation is a list of sets of nodes *Res* where each set represents a *level* of preference and maintains nodes that are equally preferable. The preference order among sets is kept by their position in the list (i.e., nodes in the set S_i are preferred to node in the set S_j if $i < j$, where i and j are the positions of the sets S_i and S_j in the list). If an expression is of the form $(e^{\succ})^*$, that is, it includes the Kleene * operator, Evaluate calls the function Closure. If this is not the case then the expression is handled by calling the function Base. The function Base is called for each set S in the input I. The two main functions by which the order among results is maintained are:

- $append\&Remove(L_1, L_2)$: it *appends* at the end of the first list L_1 elements (i.e., sets) of the second list L_2 by *removing* nodes that already appear in some set of L_1 (i.e., at a higher level of preference). If $L_1 = [S_1, ..., S_n]$ and $L_2 = [T_1, ..., T_m]$, the resulting list is $[S_1, ..., S_n, T_1 \setminus N, ..., T_m \setminus N]$, where $N = \bigcup_{i=1}^{n} S_i$.
- $append\&Merge(L_1, L_2)$: similar to the previous one with the difference that the first set of the second list is merged with the last set of the first list. If $L_1 = [S_1, ..., S_n]$ and $L_2 = [T_1, ..., T_m]$, the resulting list is $[S_1, ..., S_n \cup (T_1 \setminus N), ..., T_m \setminus N]$, where $N = \bigcup_{i=1}^{n} S_i$.

Algorithm 1 Evaluate

```
1: function EVALUATE(list of sets of nodes I, GuLP expression e^≻, graph G)
2: Output: list of sets of nodes Res
3:     Res := [];
4:     if e^≻ = (e_1^≻)* then /* contains the kleene * operator */
5:         return CLOSURE(I, e_1^≻, G, Res);
6:     for each set S in I do
7:         Res := append&Remove(Res, BASE(S, e^≻, G));
8:     return Res;
```

Algorithm 2 Closure

```
1: function CLOSURE(list of sets of nodes I,GuLP expression e^≻, graph G, list of sets of nodes Res)
2: Output: list of sets of nodes Res
3:     for i = 1 to I.size do
4:         S := remove(I.get(i), Res); /* remove from the i-th set in I nodes in Res */
5:         if i=1 then
6:             Res = append&Merge(Res, S);
7:         else
8:             Res = append&Remove(Res, S);
9:         S := EVALUATE(S, e^≻, G);
10:        S := remove(S, Res);
11:        if S.size ≠ 0 then
12:            Res = CLOSURE(S, e^≻, G, Res);
13:    return Res;
```

Algorithm 3 Base

```
1: function BASE(set of nodes nodes, GuLP expression e^≻, graph G)
2: Output: list of sets of nodes Res
3:     T := [nodes];
4:     if e^≻ =≻ (e_1^≻, ..., e_n^≻) then
5:         Res := EVALUATE(T, e_1^≻, G);
6:         for i = 2 → n do
7:             Res := append&Remove(Res, EVALUATE(T, e_i^≻, G));
8:         return Res;
9:     if e^≻ = e_1|e_2 then /* recall no preferences with union | */
10:        res_1 := EVALUATE(T, e_1, G);
11:        res_2 := EVALUATE(T, e_2, G);
12:        return [res_1.get(1) ⋃ res_2.get(1)];
13:    if e^≻ = e_1^≻/e_2^≻ then
14:        res_1 := EVALUATE(T, e_1^≻, G);
15:        return EVALUATE(res_1, e_2^≻, G);
16:    if e^≻ = e_1^≻[e_2] then /* nested expression */
17:        Res := EVALUATE(T, e_1^≻, G);
18:        for all n ∈ Res do
19:            if EVALUATE([{n}], e_2, G) has no results then
20:                remove(Res, n)
21:        return Res;
22:    if e^≻ = e_1^≻.{test} then /* test on node attributes */
23:        Res := EVALUATE(T, e_1^≻, G);
24:        test&Prune(Res, test);
25:        return Res;
26:    if e^≻ = a or e^≻ = a{test} then /* test on edge attributes */
27:        return [{x : ∃n ∈ nodes | e=(n, a, x) ∈ G ∧ checkT(e, test)}];
28:    if e^≻ = a^-1 or e^≻ = a^-1{test} then
29:        return [{x : ∃n ∈ nodes | e=(n, a, x) ∈ G ∧ checkT(e, test)}];
```

The basic functions used by BASE and CLOSURE are:

- $test\&Prune(L, test)$: remove from L all the nodes n that belong to some set $S \in L$ for which $test(n) = false$.
- $remove(S, L)$: remove from the set S all the nodes n that belong to some set $S' \in L$.
- $remove(L, n)$: remove from the list L the node n if it belongs to some set $S \in L$.

The evaluation of $e^{\succ}$ occurs from the highest preference level to the lowest. When evaluating an expression $e^{\succ}$ it is necessary to avoid to evaluate the same sub-expression multiple times starting form the same node. Hence, for each expression $e_i^{\succ}$ evaluated from a node n, the algorithm associates to the pair $\langle n, e_i^{\succ} \rangle$ the preference level p at which n has been reached from a possible previous step in the evaluation. This way, if n is reached multiple times for the same or a lower preference level, $e_i^{\succ}$ will not be evaluated again. This is because, nodes reachable from n according to $e_i^{\succ}$ have already been considered at a higher preference level.

4.1 Complexity

We focus on the following two problems.

Problem 1: PREFERENCE CHECKING
Input: An attributed graph $\mathcal{G}$, a GuLP expression $e^{\succ}$, two pairs (n, a), (n, b)
Question: Is $(n, a) \succ (n, b)$?
Output: Yes or No

The PREFERENCE CHECKING problem takes in input two pairs of nodes, where n is the starting node, and checks whether the first is preferred to the second. Note that it makes sense to define such a problem only if considering the same starting node for both pairs. The second problem concerns the computation of the actual solution.

Problem 2: PREFERENCE COMPUTATION
Input: An attributed graph $\mathcal{G}$, a GuLP expression $e^{\succ}$ a starting node n
Output: List of set of nodes $\{S_1, S_2, ...S_k\}$ s.t. $\forall\ n_1 \in S_i,\ n_2 \in S_j : n_1 \succ n_2$ if $i < j$ and $n_1 \sim n_2$ if $i = j$

Here, the problem is to compute a list of sets of nodes, where each set contains nodes reachable at the same level of preference from a starting node n given a GuLP expression $e^{\succ}$. In the following we prove that both problems can be efficiently solved. We assume that the graph $\mathcal{G}$ is stored by its adjacency list; for each node n a list of pairs (edg, n') is maintained, where edg is an edge id and n' is the node reachable by navigating edg. Given a node id its adjacency list can be accessed in time $O(1)$. To make efficient the navigation of edges in the reverse direction (i.e., l^{-1} in Table 1), an additional data structure is maintained. Edge labels and attributes are maintained in a separate data structure that can be accessed in constant time through the edge id. Same reasoning applies for node attributes. Such an approach maintains the graph topology and the attributes separate. This is useful to leverage a secondary memory data structure (e.g., an index) to maintain attributes.

Let $|\mathcal{G}|$ be the size of the adjacency list representation of the graph $\mathcal{G}$ plus the size of the tables for storing attributes (note that $|\mathcal{G}| \geq |\mathcal{N}|$, where $|\mathcal{N}|$ is the number of nodes). Let $|e^{\succ}|$ be the size of a GuLP expression. Both the PREFERENCE CHECKING and the PREFERENCE COMPUTATION problem can be solved in polynomial time in the size of the attributed graph $\mathcal{G}$ and GuLP expression $e^{\succ}$. We have:

THEOREM 4.1. *The function* EVALUATE *solves* PREFERENCE COMPUTATION *in polynomial time in* $|\mathcal{G}|$ *and* $|e^{\succ}|$.

PROOF. *(Sketch).* Let L be the list of sets of nodes produced by EVALUATE. We first sketch the proof of the correctness of the algorithm. We start with priority between nodes n_i and n_j in the same set. Recall that nodes in the same set belong to the same preference level. This is because n_i and n_j are reachable from the starting node via two paths p_i and p_j for which tests on node/edge attributes

are satisfied. Moreover, either p_i and p_j spell two strings that are equal or the two strings belong to the union of languages generated via the operator |. We now discuss priority between sets in L. Sets of nodes are added to L by EVALUATE, BASE and CLOSURE by *append&Remove* and *append&Merge*; these functions always append new sets to the end of the list by first deleting duplicates. $e^{\succ}$ is evaluated from left to right and subexpressions appearing to the left are always preferred to those appearing to the right. Hence, *append&Remove* and *append&Merge* maintain the total order of the results. We have now to show that procedure EVALUATE runs in polynomial time.

CASE 1: PREFERENCE AND NESTING FREE. This is the case in which $e^{\succ}$ is a standard regular expression plus tests on attributes. Since there are no preferences, the result consists in one set of nodes, that is, nodes reachable from a *seed* node n via paths belonging to the language of the expression for which tests on node and edge attributes are satisfied. The function EVALUATE is recursively called on all sub-expressions of $e^{\succ}$. If these sub-expressions do not contain the Kleene operator, this procedure is invoked recursively at most $O(|e^{\succ}|)$ times and the base cases (lines 22-29 of function BASE) require to check at most all the edges for all the nodes and testing attributes. This can be done in time $O(|\mathcal{G}|)$. If the Kleene operator is present, the function CLOSURE is executed at most $O(|\mathcal{N}|)$ times (if at each recursive call to CLOSURE only one node is produced in output). The EVALUATE procedure invoked from CLOSURE is recursively called at most $O(|e^{\succ}|)$ times. Since the base cases cost $O(|\mathcal{G}|)$, each call to CLOSURE costs at most $O(|\mathcal{G}| \cdot |e^{\succ}|)$ thus requiring polynomial time.

CASE 2: PREFERENCE FREE. The expression $e^{\succ}$ is a nested regular expression with tests on attributes. In this case, the procedure EVALUATE is called at line 19 of the function BASE at most $|e^{\succ}|$ times for each node. Let $e_i^{\succ}$=$e_1[e_2]$ be a sub-expression containing another nested sub-expressions (i.e,. e_2). To evaluate $e_i^{\succ}$, e_1 has to be evaluated with a cost polynomial in $|\mathcal{G}|$ and $|e_1|$. Then, for each node in the results of e_1, e_2 is evaluated in polynomial time in $|\mathcal{G}|$ and $|e_2|$.

CASE 3: FULL EXPRESSIONS. Consider the base case in which we have only one preference, that is, expressions of the form $e^{\succ}$=$\succ$ $(e_1, ..., e_k)$ (note that $e_1, ..., e_k$ are nested regular expressions). The evaluation of $e^{\succ}$ corresponds to the ordered evaluation of e_1 through e_k. The evaluation of each e_i requires polynomial time as discussed in case 1 and case 2. The size of the result of the evaluation of each e_i is at most $|\mathcal{N}|$. In order to obtain results associated to $e^{\succ}$ it is necessary to consider results from each e_i. This is done by calling k times the function *append&Remove* (line 7 function BASE). This function runs in time polynomial in $|\mathcal{N}|$ and eliminates duplicates. Hence, the evaluation of $e^{\succ}$ can be done in polynomial time and the size of the results is at most $|\mathcal{N}|$. The subsequent steps of the evaluation have to be carried out starting from at most $|\mathcal{N}|$ nodes. The most general case is $e^{\succ}$=$\succ$ $(e_1^{\succ}, ..., e_k^{\succ})$ where each $e_i^{\succ}$ may contain preferences. Since the size of the evaluation of each $e_i^{\succ}$ contains at most $|\mathcal{N}|$ nodes and can be done in polynomial time, with the above reasoning also $e^{\succ}$=$\succ$ $(e_1^{\succ}, ..., e_k^{\succ})$ can be evaluated in polynomial time and the size of the results is at most $|\mathcal{N}|$ (recall the *append&Remove* function). The evaluation of $e^{\succ}$ will produce a list $L = \{S_1, ..., S_m\}$ of disjoint sets of nodes containing at most $|\mathcal{N}|$ nodes. □

THEOREM 4.2. *The* PREFERENCE CHECKING *problem can be solved in polynomial time in* $|\mathcal{G}|$ *and* $|e^{\succ}|$.

PROOF. It is necessary to check that both a and b are contained in the result L obtained by solving the PREFERENCE CHECKING problem and that $a \succ b$. First it is necessary to identify the two sets $S_i \in L$ an $S_j \in L$ such that $a \in S_i$ and $b \in S_j$. If one of the two sets does not exist or if $i \geq j$ the answer if No, otherwise it is Yes. Since L contains at most $|\mathcal{N}|$ nodes, this checking costs at most $O(|\mathcal{N}|)$. □

5. EVALUATION

GULP-IT is a system implemented in Java for evaluating GULP expressions and is available at the GULP Web site [1].

Evaluation methodology. We performed three kinds of experiments. The first aims at showing that adding the preference operator does not incur in any increasing in the time of the evaluation of GULP expressions. The second experiment advocates the need of having the preference operator *inside* GULP. The third experiment has been carried out by using a set of complex expressions with preferences. Here, the aim is to show the advantages of using a graph query language with preferences in order to differentiate among the nodes associated to an answer. All the experiments have been performed on an Intel Core i5 2.4 GHz with 4GBs of memory. Moreover, each experiment has been executed 5 times and the average time has been considered.

Table 3: Description of the datasets.

Dataset	#Nodes	#Edges	#Node attr. val.	#Edge attr. val.
$DBLP_F$	1.2M	2.5M	2.5M	1.4M
$DBLP_S$	485K	860K	970K	1M

Datasets. In order to test GULP-IT in a real context, we used the Proximity DBLP dataset, which is available at `http://kdl.cs.umass.edu/data/dblp/dblp-info.html`. The dataset is a snapshot of DBLP as of April 12, 2006 and includes six different types of publications and links from publications to their authors and editors and from papers to journals, proceedings, or books in which they appear. Citation links from one publication to another are also included. Interestingly, this dataset provides both node and edge attributes. To test the scalability of GULP-IT, we considered two datasets. The first is a subset of Proximity DBLP, referred to as $DBLP_S$ where only journals, journal papers and their authors have been considered. The second one is the full dataset and is referred to as $DBLP_F$. Table 3 summarizes the datasets. Due to the magnitude of the graph, attributes are kept in an index stored in secondary memory.

Experiment 1. We performed some experiments in $DBLP_F$ to test the cost of performing requests with preferences. We compared two queries with the following structure:

1. *seed* (`author-of[in-journal])|(author-of[in-proc]`)
2. *seed* $\succ$(`author-of[in-journal], author-of[in-proc]`)

where *seed* is a starting node in the graph. The first, prototype is a query involving the union of all the authors of journal papers reachable from the seed node with all the authors of papers in proceedings. On the other hand, the

[1] `http://graphgulp.wordpress.com`

second prototype involves preferences. In particular, journal papers are preferred to papers in proceedings. We tested four queries by using as seed node each time a different computer scientist. The result of the evaluation is reported in Fig. 3. As it can be observed, using the preference operator $\succ$ does not bring any additional cost as compared to the case in which the union is used. Times differ from query to query due to the different number of nodes reachable from the seed. For instance, **Q4** for which *Christos Papadimitriou* has been used as seed node, took longer since the total number of publications was higher than in the other cases. In particular, 128 publications were found in the first preference class (i.e., Journals) while 108 in the second (i.e., proceedings). Note that with the union, it would not have been possible to distinguish between the classes; one would have gotten all the nodes in a single set.

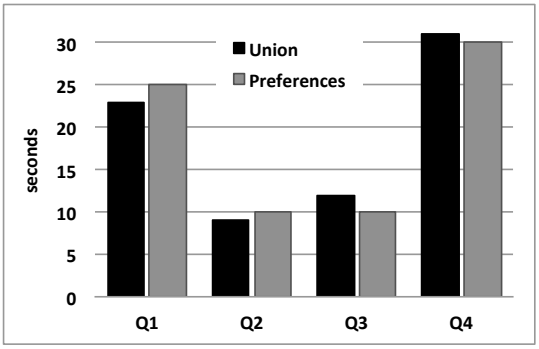


Figure 3: Union vs. preference.

Another experiment has been performed on $DBLP_S$ by using the expression $\succ$(`in-journal`$^{-1}$`{@in_year<=`K`}, in-journal`$^{-1}$`{@in_year>`K`)}` with K ranging from 1996 to 2004 and considering even years. This gives a total of 5 query templates. Note that here no seed node has been specified so the query has been executed starting from each of the 545 nodes representing journals in the dataset. Each query defines two classes of preference: the first for journal publications before K while the second one for publications after K. On the average a query took 150s. for a total of around 270K publications found with an average of 496 papers per journal. For each query and for each K, we computed the average number of preference classes P_n over all the 545 journals. This can be seen as an indicator of how results can be partitioned in two classes. The more P_n approaches 2 the more the result (i.e., papers in journals) can be partitioned in preference classes. For $K = 2002$ we obtained the maximum value of $P_n = 1.68$.

Experiment 2. This experiment is motivated by the following consideration. An expression with preferences $e^{\succ}$ can be decomposed into a set of expressions without preferences $S_{e^{\succ}}$. If we denote with Q the number of times the preference operator $\succ$ is used in $e^{\succ}$ and with T the number of terms in each $e^{\succ}$, then the total number of expressions is T^Q. For instance evaluating $e^{\succ} = \succ (e_1, e_2)/p_4$ is conceptually equivalent to the ordered evaluation of the two expressions e_1/p_4 and e_2/p_4, where each evaluation produces a set (i.e., a preference class). A naive approach for evaluating $e^{\succ}$ would construct the corresponding set of expressions without preferences and evaluate them one by one. We compared the behavior of GuLP-it with the naive approach over a synthetic dataset consisting of graphs with size ranging from 100 to 100K nodes. Each graph is obtained starting from a set of nodes with a ring topology and two types of edges (p_1 and p_2); each node is connected to both the predecessor and successor with edges labeled p_1 and p_2. For each of such graphs the expression used has the following shape $e^{\succ} = \succ (p_1, p_2)/ \succ (p_1, p_2)...$ with the number of preferences (#pref) ranging from 1 to 10. For instance #pref=3 corresponds to the expression $e^{\succ} = \succ (p_1, p_2)/ \succ (p_1, p_2)/ \succ (p_1, p_2)$. Results are reported in Fig. 4.

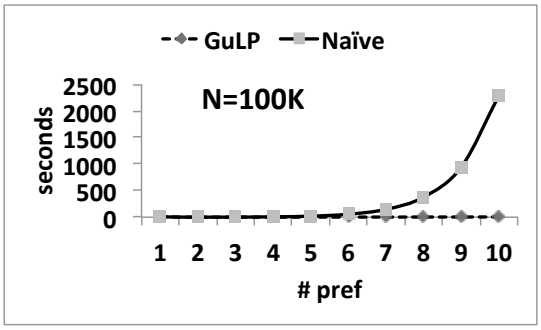


Figure 4: Naive vs. GuLP-it **varying preferences.**

As it can be noted, the execution time for the naive approach grows exponentially with the number of preferences and size of the graph. On the other hand, GuLP-it shows a polynomial behavior. As an example, while with the naive approach it took around 2400s to evaluate an expression with 10 preferences, with GuLP-it it took around 50s. Here we report the results for $N = 100K$. Similar trends have been observed for $N = 100, N = 1000, N = 10K$.

We conduct an additional set of experiments by keeping constant the number of preferences #pref to 3, 5 and 10 and varying the number of nodes. Even in this case, an exponential behavior has been observed for the naive approach while GuLP-it still maintains a low running time (results for #pref=5 are reported in Fig. 5). The motivation underlying such a different behavior of the naive approach w.r.t. GuLP-it is the following. With the naive approach for each expression, obtained from the original expression with preferences, the computation starts from scratch and so all nodes can possibly be part of the results. On the other hand, GuLP-it is able to discard nodes considered during the evaluation at a higher preference level thus reducing the number of possible paths to be explored in order to build the result set. This underlines the need to have the preference operator embedded in the language and the algorithm discussed in Section 4.

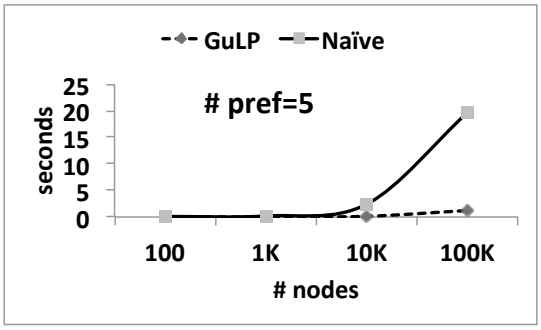
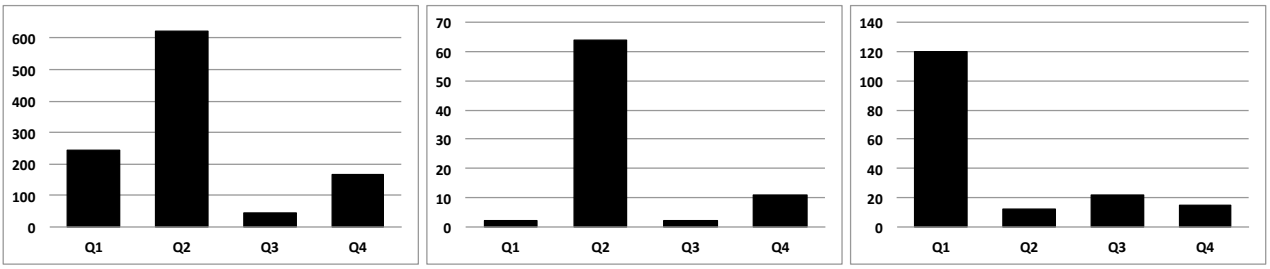


Figure 5: Naive vs. GuLP-it **varying** N.

Table 4: Query set for experiment 3.

Q1 = $\succ$ ((author-of[in-journal{in-year<=1970}]/author-of^{-1})*,(author-of[in-journal{in-year>1970∧in-year<=1975}]/author-of^{-1})*)
Q2 = ($\succ$((author-of[in-journal{in-year<=1970}]/author-of^{-1}),(author-of[in-journal{in-year>1970∧in-year<=1975}]/author-of^{-1})))*

Q3 = $\succ$((author-of[in-journal{in-year<=1970}]/author-of^{-1})*,(author-of[in-proc.{year<=1970}]/author-of^{-1})*)
Q4 = ($\succ$((author-of[in-journal{in-year<=1970}]/author-of^{-1}),(author-of[in-proc.{year<=1970}]/author-of^{-1})))*



(a) Co-authorship net size (b) #Preference classes (c) AVG # nodes per preference class

Figure 6: Evaluation of GuLP in Experiment 3

Experiment 3. The third experiment uses a set of complex queries (see Table 4) involving nesting, preferences on edges, paths and tests over node and/or edge attributes. Queries $Q1$ and $Q2$ have been executed on the dataset $DBLP_S$ while $Q3$ and $Q4$ on $DBLP_F$. All the 4 queries leverage co-authorship relations. In general, co-authorship between persons holds if there is at least one paper in common. GuLP enables to express this via the expression `author-of/author-of`$^{-1}$. This is because, from a person via the edge `author-of` it is possible to reach paper nodes. Then, with `author-of`$^{-1}$ it is possible to navigate in the reverse direction from paper nodes toward all their authors. In $Q1$ and $Q2$ two ad-hoc definitions of co-authorship are expressed. The first applies only if two persons have in common at least one paper published before or in 1970. This is expressed in GuLP via co_1=`author-of[in-journal{in-year<=1970}]/author-of`$^{-1}$. The expression within `[ ]` selects among all papers reached via `author-of` only those published in journals before or in 1970. This filtering is also performed by a test over the edge attribute `in-year`. The second definition of co-authorship is expressed by co_2=`author-of[in-journal{in-year>1970, in-year<=1975}]/author-of`$^{-1}$ and holds only if there are common papers published between 1970 and 1975. The aim of $Q1$ and $Q2$ is to compute the *closure* over co_1 and co_2. This corresponds to identify all persons for which there exists a co-author path of any length, starting from a seed node n. To simplify the presentation, we can rewrite $Q1$ and $Q2$ according to co_1 and co_2. In particular $Q1 = \succ (co_1^*, co_2^*)$ while $Q2 = (\succ (co_1, co_2))^*$. $Q1$ defines two classes of preference. The first, corresponding to co_1^*, includes all persons reachable from the seed node via paths of whatever length defined over co-authorship relations complying with co_1. The second, corresponding to co_2^*, is defined similarly. According to the semantics of GuLP expressions, the results of the evaluation of $Q1$ and $Q2$ are the lists of sets L_{Q1} and L_{Q2} (see Section 3.1) defined as:

$$L_{Q1} = [[\![co_1^*]\!]^{\mathbf{n}}_{\mathcal{G}<0>}, [\![co_2^*]\!]^{\mathbf{n}}_{\mathcal{G}<0>}] \quad (1)$$

$$L_{Q2} = [[\![co_1^*]\!]^{\mathbf{n}}_{\mathcal{G}<0>}, [\![co_1^*/co_2]\!]^{\mathbf{n}}_{\mathcal{G}<0>}, [\![co_1^*/co_2/co_2]\!]^{\mathbf{n}}_{\mathcal{G}<0>}, ..., [\![co_2/co_1^*]\!]^{\mathbf{n}}_{\mathcal{G}<0>}, ..., [\![co_2^*]\!]^{\mathbf{n}}_{\mathcal{G}<0>}] \quad (2)$$

As it can be noted, the first preference class in L_{Q1} is the same as the first preference class in L_{Q2}. As for the second class in L_{Q1} and the last class in L_{Q2}, although being defined through the same expression, they may contain different results. This is because in L_{Q2} there are other preference classes at a higher level of preference (i.e., they have a lower index in the list) that may contain some of the nodes contained the last preference class of L_{Q1}. Recall, that each node must belong to at most one preference class.

$Q3$ and $Q4$, which have been evaluated over the full DBLP dataset, have a similar shape to $Q1$ and $Q2$, respectively. Even here, two definitions of co-authorship have been used. The first is the same as co_1 discussed before. The second one co_3=`author-of[in-proc.{year<=1970}]/author-of`$^{-1}$ considers papers in proceedings and published before or in 1970 and performs tests on the node attribute `year`. All queries have been executed using *Jeffrey D. Ullman* as the seed node. Fig. 6 reports the results for Experiment 3. In particular, for each query, the co-authorship network size, the number of preference classes and the average number of co-authors per preference class is reported. As it can be noted in Fig. 6 (a), $Q2$ enables discovering the largest network of co-authors. Moreover, $Q2$ provided more results than $Q1$. This is because, $Q2$ thanks to the Kleene operator outside the preference (see Table 3) enables specifying a larger number of kinds of co-authorship paths than $Q1$ (see eq.1 and eq. 2). Hence, $Q2$ provides also a larger number of preference classes (see Fig. 6 (b)). The same reasoning applies for $Q4$ w.r.t. $Q3$. Fig. 6 (c) reports the average number of nodes per preference class.

Fig. 7 reports for each query the distribution of co-authors in the various preference classes. For instance, $Q1$ has 28 persons in the first preference class, defined by co_1^* and 214 in the second one defined by co_2^*. On the average the execution time for $Q1$ and $Q2$ over $DBLP_S$ was of approximatively 15min. For $Q3$ and $Q4$ executed over the larger $DBLP_F$ average the time was of 3.5 min.

6. CONCLUDING REMARKS

We described GuLP, a graph query language to express preferences over node/edges attributes and complex paths. We showed that the evaluation of expressions with preferences can be done in polynomial time as in the case of a language without preferences. The embedding of the operator $\succ$ in the language is useful since a naive strategy rewriting

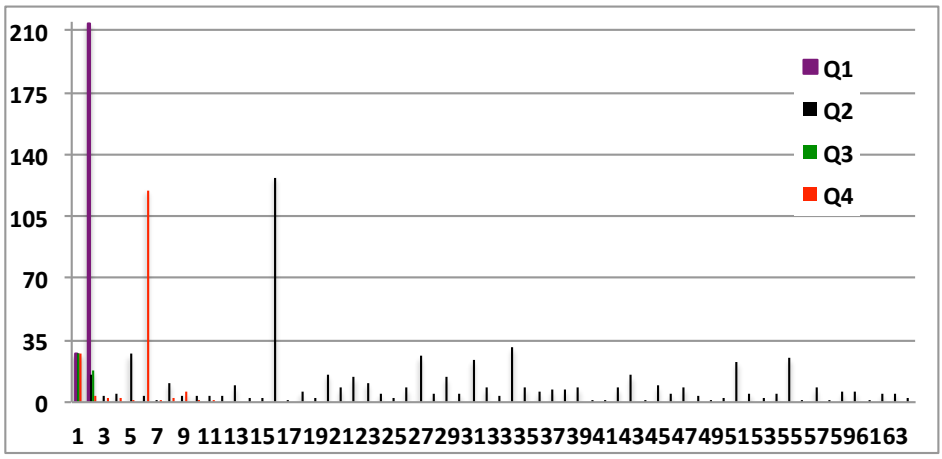


Figure 7: # co-authors per preference class

an expression with preferences in a set of standard expressions (to be executed in order) has an exponential running time. We did not focus on the orthogonal problem of learning preferences [18], which will be addressed in the future.

Our framework adopts a boolean matching model: nodes along the path are considered if they match a given attribute/edge value. An interesting extension is that of combining this approach with a fuzzy matching operator over attribute values [9]. From a more theoretical point of view, an interesting line of future research is the comparison of the $\succ$ operator, which can be seen as an n-ary relation over paths (that are not explicitly retrieved) and expressive classes of languages such as Extended Regular Path Queries that look into regular and rational relations [4, 3]. Considering other data models (e.g., RDF) is also in our research agenda.

7. REFERENCES

[1] S. Amer-Yahia, I. Fundulaki, and L. Lakshmanan. Personalizing XML Search in Pimento. In *ICDE*, pages 906–915. IEEE, 2007.

[2] R. Angles and C. Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys*, 40(1):1–39, 2 2008.

[3] P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations and the generalized intersection problem. In *LICS*, pages 115–124, 2012.

[4] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive Languages for Path Queries over Graph-structured Data. In *PODS*, pages 3–14. ACM, 2010.

[5] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data – The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[6] J. Chomicki. Querying with Intrinsic Preferences. In *EDBT*, LNCS, pages 34–51. Springer, 2002.

[7] J. Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems*, 28(4):427–466, 2003.

[8] C. Domshlak, E. Hüllermeier, S. Kaci, and H. Prade. Preferences in AI: An overview. *Artificial Intelligence*, 175(7):1037–1052, 2011.

[9] R. Fagin. Combining Fuzzy Information from Multiple Systems. *Journal of Computer System Sciences*, 58(1):83–99, 1999.

[10] R. Fagin and E. L. Wimmers. Incorporating User Preferences in Multimedia Queries. In *ICDT*, LNCS, pages 247–261. Springer, 1997.

[11] V. Fionda, C. Gutierrez, and G. Pirró. Semantic Navigation on the Web of Data: Specification of Routes, Web Fragments and Actions. In *WWW*, pages 281–290. ACM, 2012.

[12] S. Flesca and S. Greco. Partially Ordered Regular Languages for Graph Queries. *Journal of Computer System Sciences*, 70(1):1–25, 2005.

[13] G. Grahne, A. Thomo, and W. Wadge. Preferential Regular Path Queries. *Fundamenta Informaticae*, 89(2-3):259–288, 2008.

[14] W. Kießling. Foundations of Peferences in Database Systems. In *VLDB*, pages 311–322, 2002.

[15] W. Kießling, B. Hafenrichter, S. Fischer, and S. Holland. Preference XPath: A Query Language for e-commerce. In *Information Age Economy*, pages 427–440, 2001.

[16] G. Koutrika and Y. E. Ioannidis. Personalization of Queries in Database Systems. In *ICDE*, pages 597–608. IEEE, 2004.

[17] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010.

[18] K. Stefanidis, M. Drosou, and E. Pitoura. Perk: Personalized Keyword Search in Relational Databases through Preferences. In *EDBT*, pages 585–596. ACM, 2010.

[19] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Transactions on Database Systems*, 36(3):1–45, 2011.

[20] R. Torlone and P. Ciaccia. Management of User Preferences in Data Intensive Applications. In *SEBD*, pages 257–268, 2003.

[21] M.-E. Vidal, L. Raschid, and J. Mestre. Challenges in Selecting Paths for Navigational Queries: Trade-Off of Benefit of Path versus Cost of Plan. In *WebDB*, pages 61–66, 2004.

[22] P. T. Wood. Query Languages for Graph Databases. *SIGMOD Record*, 41(1):50–60, 2012.