Regular Path Queries in Graph Databases

Lewis Dyer

March 29, 2023

- Formally define Regular Path Queries, and give an example or two.
- Formally introduce the main semantics of regular path queries, defining the main algorithmic questions in play.
- Give some results so far.

1 Preliminaries

As a model for graph databases, we consider directed edge-labelled graphs. Given an alphabet Σ, a graph $G = (V, E)$ consists of a set of vertices V, along with a set of edges $E \subseteq (V \times \Sigma \times V)$, with (u, l, v) denoting at edge from vertex u to vertex b with label l. Note that this model requires that every edge is labeled, and that we may omit the label if this label is irrelevant or clear from context. We further note that loops and multiple edges are permitted in this model.

Given two vertices u and v, a path p is defined by a sequence of edges of the form $(v_0, l_1, v_1), (v_1, l_2, v_2), \ldots, (v_{n-1}, l_n, v_n)$ where $u = v_0$ and $v = v_1$, and we say this path has length n. We denote $\mathcal{P}_(G, u, v)$ as the set of all such paths in G between u and v, occasionally omitting the subscript where this is clear from context. We say a path is *simple* if any two vertices v_i, v_j in p are pairwise distinct.

Given an alphabet Σ, we define a regular expression r as follows. First, the empty string ϵ is a regular expression, along with any character a with $a \in \Sigma$. Then, given two regular expressions r and s, their concatenation (rs), denoting the set of strings obtained by concatenating a string accepted in r and a string accepted by s, is also a regular expression. Their alternation $(r|s)$, denoting the set of strings accepted by r or s, is also a regular expression, along with (r^*), the set of finite strings obtained by concatenating zero or more strings which are recognised by r. We also allow for ommitting parentheses where there is no ambiguity. We say the *language* of r, denoted $L(r) \subset \Sigma^*$, is the set of strings which are accepted by r.

Regular expressions can also be described using nondeterministic finite automata (or an NFA for short), which are tuples of the form $N = (Q, \Sigma, \Delta, Q_0, Q_1)$. Here Q is a finite set of states in the automata, Σ is a finite alphabet as for

regular expressions, Q_0 is a set of initial states in Q, Q_1 is a set of accepting states in Q, and $\Delta \subseteq Q \times \Sigma \times Q$ representing transitions between states, such that (q, a, q') represents a transition from q to q' via the character a. Note that nondeterminism here means that a single state may have multiple transitions to different states using the same character. We may also define the product of a graph G with start and end vertices s and t and a nondeterministic finite automata N, which is a graph $G \times N$ such that $V(G \times N) = V \times Q$ and $E(G \times N) = \{((u_1, q_1), a, (u_2, q_2))|(u_1, a, u_2) \in E \text{ and } (q_1, a, q_2) \in Delta.$ In other words, an edge with label a exists between two vertex-state pairs if and only if an edge with label a exists between both vertices in G, and moreover there exists a transition in N between both states with label a. We further note that this may also be treated as an NFA by treating the sets of vertices and edges as the set of states and transitions respectively, and by defining $Q_0 = \{(s, q)|q \in Q\}$ and $Q_1 = \{(t, q)|q \in Q\}$.

2 Introducing regular path queries

Given a path p in a graph G and vertices u and v, we consider the word w_p formed in Σ^* by concatenating each edge label in p. More formally, the function $w : \mathcal{P}_{(G,u,v)} \to \Sigma^*$ with $w\left((v_0, l_1, v_1), (v_1, l_2, v_2), \ldots, (v_{n-1}, l_n, v_n)\right) = l_1 l_2 \ldots l_n$ produces the unique word associated with a given path p. Regular path queries (also denoted as RPQs) consist of a regular expression r along with a set of *semantics* for that RPQ, though we frequently refer to an RPQ solely by the regular expression where the semantics being used are clear from context. Evaluating an RPQ over a graph G between vertices u and v consists of two steps: first, we consider paths $p \in \mathcal{P}_{(G,u,v)}$ such that $w(p)$ is accepted by r, and we denote these paths as the set of *candidate paths* of the RPQ. Then, we determine which of these candidate paths to output depending on the semantics being used. Three typical sets of semantics are:

- **Arbitrary semantics**, where all candidate paths are returned.
- **Shortest path semantics**, where all candidate paths of minimum length are returned.
- **Simple path semantics**, where all simple candidate paths are returned.

Each of these sets of semantics presents different challenges. When using arbitrary semantics, care must be taken to avoid paths of infinite length, which can occur in cyclic directed graphs. For example, evaluating a^* over the graph in Figure 2 from vertex x to vertex y using arbitrary semantics will result in an infinite set of paths with words given by the set $\{a^{3k+1}|k \in \mathbb{N}\}$.

Shortest path semantics avoid this problem, but can be counter-intuitive even for common queries. For example, when counting the number of paths between two vertices x and y that are accepted by some regular expression r, adding an additional path between x and y can actually reduce the overall count, if this newly added path is accepted by r and is shorter than any other paths.

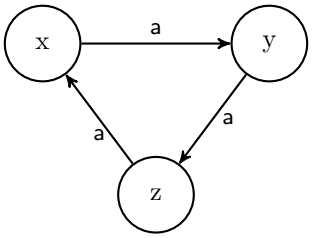

Figure 1: A directed cyclic graph on 3 vertices, where every edge has label a.

Simple paths avoid this lack of intuition while also avoiding paths of infinite length, and correspond to many natural constraints on paths in common queries. For example, in a transport network where vertices correspond to locations and edges correspond to different forms of travel between locations, finding simple paths between two locations means that visiting the same location more than once is not permitted.

Regular path queries are applied in SPARQL, a database query language providing specific syntax for graph traversals, via the use of *property paths* [1]. For example, the following query from the Wikidata knowledge graph [2] uses such a property path:

```
SELECT ?siteLabel ?coord ?image ?site
WHERE {
    ?site wdt:P31/wdt:P279* wd:Q839954 ;
        wdt:P625 ?coord ;
        wdt:P18 ?image .
SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
}
```

This query returns a list of all archaeological sites in the knowledge graph, along with their names, locations and an associated image. Most notably, `?site wdt:P31/wdt:P279* wd:Q839954` represents a regular path query between the vertex `?site`, representing a candidate instance of an archaeological site, and the vertex `wd:Q839954` representing the concept of an archaeological site, where any desired path consists of a sequence of edges with edge labels either `wdt:P31`, representing that an entity is an instance of a concept, or `wdt:P279` representing that one concept is a subclass of another concept. The need for a property path here arises because specific archaeological sites may not be adjacent to vertex `wd:Q839954`, and may be instances of more specific concepts instead. For example, Duncarnock Hill Fort is an instance of a contour fort, which is a subclass of hillforts which are in turn a subclass of ancient monuments, which are finally subclasses of archaeological sites. Hence, regular path queries are especially suited to knowledge graphs, or any sort of graph database with hierachical relationships between entities.

3 Complexity of evaluating RPQs

To begin, we first formalise the set of evaluation problems on RPQs we are considering. We define the decision RPQ evaluation problem with arbitrary path semantics as follows:

> PATH
> **Input:** A graph G with a starting vertex s and an ending vertex e, and a regular expression r.
>
> **Output:** `true` if there exists a path p in G matching s and r such that $w(p)$ is accepted by r, and `false` otherwise.

Note that this defines the decision problem for RPQ evaluation under arbitrary path semantics. Analogous variants of this decision problem exist for the other forms of semantics introduced in Section 2, which we shall denote as ShortPath and SimPath for shortest path and simple path semantics respectively. Similarly, we may also define enumeration variants of these problems, where all such paths from s to r whose words are accepted by r should be returned. We denote these problems as EnumPaths, EnumShortPaths and EnumSimPaths.

Firstly, we know that Path is solvable in polynomial time for any regular expression r. This follows by taking the product of G and the NFA N representing the regular expression r, and determining if there exists a path from (s, q_0) to (t, q_1), for some $q_0 \in Q_0$ and $q_1 \in Q_1$.

Considering this product as an NFA allows us to extend this procedure to solve EnumPaths in polynomial delay. Specifically, we produce a new NFA N' by replacing all transitions of the form $((u_1, q_1), a, (u_2, q_2))$ with $((u_1, q_1), (u_1, a, u_2), (u_2, q_2))$, such that the set of labels Σ now represents the set of transitions in N. Hence, enumerating paths from s to t such that their words are accepted by r is equivalent to enumerating every word that is accepted by N', which Ackerman and Shallit show in [3] can be computed in polynomial delay. We further have that EnumShortPaths can also be computed in polynomial delay - since the algorithm introduced by Ackerman and Shallit returns words in radix order, we may simply just terminate when the length of a word accepted by r is enumerated that is longer than the previous enumerated word.

4 Characterising RPQs for simple path semantics

As discussed in Section 3, SimPath and EnumSimPaths are in general intractable over all possible regular expressions, with SimPath being NP-complete with the regular expression $(aa)^*$ as shown by Lapaugh and Papadimitriou in [4], and EnumSimPaths being #P-complete with the regular expression a^* as shown by Valiant in [5]. A natural further question is whether a particular class of regular expressions allows for tractable decision and enumeration, and more-

over whether this class is sufficiently powerful to handle regular path queries that are used in practice.

Martens and Trauter tackle this question in [6] by defining the class of *simple transitive expressions* (STEs for short). In this class, we define *atomic expressions* as expressions of the form $(a_1 + \dots a_n)$ for $a_i \in \Sigma$, which we denote as A. Note that since the order of elements in this expression is unimportant, we also denote A as a subset of Σ, such that $a_i \in A$ if and only if a_i is accepted by A, and we also allow for A to be the emptyset, in which case ϵ is the only string accepted by A. We then further define *bounded expressions* as expressions of the form $A_1 \dots A_k$ or $A_1? \dots A_k?$ for some $k \geq 0$ such that each A_i is an atomic expression. Finally, a simple transitive expression is an expression of the form $B_{pre}A^*B_{post}$, such that B_{pre} and B_{post} are bounded expressions while A is an atomic expression.

Intuitively, the main idea behind STEs is that a bounded amount of local navigation is permitted at the start and end of each expression, consisting of testing paths either of length exactly k or with length at most k. Then, the optional transitive step in the middle of the expression is sufficiently simple.

Following this, Martens and Trauter define a dichotomy over classes of STEs using the notion of *cuttability* - intuitively speaking, that parts of the expression which do not match the transitive part should only occur within a constant distance of the start or end of the expression. More precisely, we define the *left cut border* of an STE r as the largest value c_l such that, for $B_pre = A_1 \dots A_k$, T is not a subset of A_{c_l}, or zero if either this value does not exist or B_pre is of the form $A_1? \dots A_k?$. The right cut border c_r is defined analogously, reversing the indices in B_{post}, and we say an expression has cut border c if the maximum of its left and right cut borders is c, and in particular that a class of STEs $\mathcal{R}$ is *cuttable* if there exists a constant c such that every expression in $\mathcal{R}$ has cut border at most c. We also need one more mild condition on classes of STEs. Given some class of STEs $\mathcal{R}$, we say $\mathcal{R}$ can be sampled if there exists an algorithm that, given any k, either returns a expression in $\mathcal{R}$ whose cut border is greater than k, or else returns that such an expression does not exist.

Given this relatively mild condition, we obtain the following dichotomy: for any class of STEs $\mathcal{R}$ that can be sampled, if $\mathcal{R}$ is cuttable then SimPath is FPT over any $r \in \mathcal{R}$ parameterised by the length of r. Otherwise, SimPath is W[1]-hard for any $r \in \mathcal{R}$.

While this dichotomy is powerful within the class of STEs, one notable shortcoming is that the class of STEs is quite a limited subset of regular expressions. For example, the regular expressions $a + b^+$ and $(ab)*$ are not STEs. However, analysis of SPARQL query logs performed by Bonifati et al. in [7] shows that 99.992% of regular path queries in these query logs are STEs, and the vast majority of STEs in these query logs have cut border at most 2.

Moving onto enumeration problems, Yen's algorithm [8] provides a polyomial delay algorithm to enumerate all simple paths between two vertices s and t, without any sort of labelling on edges. Informally, this algorithm operates by finding a shortest path between s and t, since any shortest path must also be a simple path. We then consider modifications of this path where the first i

edges coincide match but the $(i+1)$th edge differs, continually repeating this procedure until all simple paths are found.

Turning back to edge labels, Martens and Trauter show this algorithm can be modified to to solve EnumSimPaths in polynomial delay for the set of regular expressions that are downward closed, which are the set of regular expressions r such that for any word $a_1 a_2 \dots a_n \in L(r)$, and any sequence $0 < i_1 < \dots < i_k < n+1$, we have that the subsequence $a_{i_1} \dots a_{i_k}$ is in $L(r)$. This result is further extended by modifying Yen's algorithm to compute simple paths rather than shortest paths, allowing for EnumSimPaths to be solved in polynomial delay for all cuttable STEs.

References

[1] "SPARQL 1.1 Query Language." https://www.w3.org/TR/sparql11-query/#propertypaths.

[2] "Wikidata:SPARQL query service/queries/examples/advanced - Wikidata." https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples/advanced#Locations_

[3] M. Ackerman and J. Shallit, "Efficient enumeration of words in regular languages," *Theoretical Computer Science*, vol. 410, pp. 3461–3470, Sept. 2009.

[4] A. S. Lapaugh and C. H. Papadimitriou, "The even-path problem for graphs and digraphs," *Networks*, vol. 14, no. 4, pp. 507–513, 1984.

[5] L. G. Valiant, "The complexity of computing the permanent," *Theoretical Computer Science*, vol. 8, pp. 189–201, Jan. 1979.

[6] W. Martens and T. Trautner, "Evaluation and Enumeration Problems for Regular Path Queries," in *21st International Conference on Database Theory (ICDT 2018)* (B. Kimelfeld and Y. Amsterdamer, eds.), vol. 98 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 19:1–19:21, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[7] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large SPARQL query logs," *The VLDB Journal*, vol. 29, pp. 655–679, May 2020.

[8] J. Y. Yen, "Finding the K Shortest Loopless Paths in a Network," *Management Science*, vol. 17, pp. 712–716, July 1971.