Structural Characterization of Graph Navigational Languages* (Extended Abstract)

Valeria Fionda[1] and Giuseppe Pirrò[2]

[1] DeMaCS, University of Calabria, Italy
fionda@mat.unical.it
[2] Institute for High Performance Computing and Networking, ICAR-CNR, Italy
pirro@icar.cnr.it

Abstract. Graph navigational languages define binary relations in terms of pair of nodes in a graph subject to the existence of a path satisfying a certain regular expression. The goal of this paper is to give a novel characterization of navigational languages in terms of the structure of the graph embracing the results of a query. We define novel graph-based query evaluation semantics and efficient algorithms able to represent and capture intermediate nodes/edges linking pairs of nodes in the answer. We enhance the language of Nested Regular Expressions (NREs) with our machineries, thus defining the language of Structural NREs (sNREs).

1 Introduction

Graph data pervade everyday's life; social networks, biological networks, and Linked Open Data on the Web are just a few examples of its spread and flexibility. The limited support that relational query languages offer in terms of recursion stimulated the design of graph query languages where *navigation* is a first-class citizen. Regular Path Queries (RPQs) and their variants allowing conjunction (CRPQs) and inverse (C2RPQs) [5], and Nested Regular Expressions (NREs) [11] are some examples. More recently, also SPARQL has been extended with a (limited) navigational core called property paths (PPs). The complexity of query evaluation for these languages is now well understood; the only tractable languages are 2RPQs and NREs while the introduction of conjunction (C2RPQs) makes the problem intractable (NP-complete) and the evaluation of PPs still suffers from some problems (mixing set and bag semantics) [2].

Queries expressible with these languages ask for *pairs of nodes* connected by a path conforming to a regular language over binary relations. As an example, Syd can find that one of his co-authors at distance 3 is Mark; nevertheless, no information about the structure (i.e., intermediate nodes/edges) of the graph where the pair (Syd, Mark) appears is given. This missing piece of information

* An extended version of this work titled "Explaining Graph Navigational Queries" will appear in proc. of the 14th Extended Semantic Web Conference (ESWC).

would allow to understand why a query has returned a certain set of (pairs of) nodes and is useful in contexts where one needs to connect the dots, such as in bibliographic networks, generic exploratory search or query debugging [6]. The broad goal of this paper is to introduce Structural Nested Regular Expressions (sNREs in short), an enhancement of Nested Regular Expressions (NREs) tackling query answering from the point of view of the structure of the graph where the results of a query appear. The challenge that we face with sNREs is to *define formal semantics and efficient algorithms that allow navigational queries to return graphs while keeping query evaluation tractable.*

Related Work. There exists a solid body of work about graph query languages. The core of such languages are Regular Path Queries (RPQs) that have been extended with other features, among which, conjunction (CRPQs) [5], inverse (C2RPQs) and the possibility to return and compare paths (EXPQs) [4]. These additions do add expressive power at the cost of making query evaluation intractable (combined complexity). Languages such as Nested Regular Expressions (NREs) [11] allow existential tests in the form of nesting, in a similar spirit to PDL and XPath while offering tractable query evaluation. Other strands of research (e.g., [10, 3]) feature more expressive languages than NREs with a higher computational cost. Our work differs from related research in one main respect: We *do not* focus on increasing the expressiveness of existing languages with new features (e.g., path variables and path variable constraints) as done with CRPQs, ECRPQs. Our goal is to provide a *novel structural characterization of graph navigational languages* taking as a yardstick the language of NREs. Such characterzation amount at studying how NREs can be enhanced to also return structural information (i.e., graphs) without hindering the complexity of query evaluation.

Contributions and Outline. We make the following main contributions: *(i)* a structural characterization of Nested Regular Expressions that we call Structural NREs (sNREs), which to the best of our knowledge is the first graph navigational language able to return graphs as the answer to a navigational query; *(ii)* formal semantics based on novel operators that work with graphs besides pairs of nodes; *(iii)* efficient algorithms for query evaluation under the novel semantics.

The remainder of the paper is organized as follows. Section 2 presents the sNREs language. Section 3 presents query evaluation algorithms and a study of their complexity. We conclude and sketch future work in Section 4.

2 Structural Nested Regular Expressions

We focus our attention on the Resource Description Framework (RDF). An RDF triple is a tuple of the form $\langle s, p, o\rangle \in \mathbf{I} \times \mathbf{I} \times \mathbf{I} \cup \mathbf{L}$, where $\mathbf{I}$ (IRIs) and $\mathbf{L}$ (literals) are countably infinite sets; for ease of exposition we do not consider blank nodes. An RDF graph G is a set of triples. The set of terms of a graph will be $terms(G) \subseteq \mathbf{I} \cup \mathbf{L}$; $nodes(G)$ will be the set of terms used as a subject or object of a triple while $triples(G)$ is the set of triples in G. Since SPARQL

property paths offer very limited expressive power we will consider the language of Nested Regular Expressions (NREs) [11] as reference language.

Nested Regular Expressions. NREs, originally proposed as the navigational core of SPARQL, allow to express existential tests along the nodes in a path via nesting (in the same spirit of XPath) while keeping the (combined) complexity of query evaluation tractable. NREs allow to specify *pairs of nodes* in a graph G, subject to the existence of a path satisfying a certain regular condition among them. Each NRE *nexp* over an alphabet of symbols Σ defines a *binary relation* $[\![nexp]\!]^G$ when evaluated over a graph G. The result of the evaluation of an NRE *is a set of pairs of nodes*. In the spirit of NREs, several extensions have been defined (e.g., EPPs [7]). Although adding expressive power to NREs (e.g., EPPs add path conjunction and (safe) negation), these languages all return pairs of nodes. This motivates the introduction of sNREs, which to the best of our knowledge is the first language tackling the problem of returning graphs from the evaluation of navigational queries while remaining tractable.

2.1 The sNREs Language

The syntax of sNREs is defined by the following grammar:

$$snrexp := \tau\#exp \quad (\tau \in \{full, filt, set\})$$
$$exp := a \mid \hat{}a \mid exp/exp \mid exp|exp \mid exp^* \mid exp[exp_1]$$

where a$\in\Sigma$, $\hat{}$ denotes backward navigation, / path concatenation, | path union, and $[exp_1]$ an existential test in the form of a nested expression. It is interesting to note that, syntactically, sNREs differs from NREs (and other navigational languages) only for the construct τ. sNREs offers a flexible way of choosing the desired output. τ allows to specify different evaluation semantics that output: *(i)* pairs of nodes (i.e., *set*) as in NREs; *(ii)* structural information in terms of the whole portion of the graph "touched" during the evaluation ($full$); *(iii)* structural information in terms of (union of) paths leading to some result ($filt$).

On the expressiveness of sNREs. We compare sNREs with NREs and other languages that deal with paths/subgraphs. As for NREs, it is easy to see that *the output* of a sNREs query, when specifying structural information via $full$ or $filt$, is richer than that of its corresponding NREs. This occurs because graph navigation is only referenced in the semantics of NREs *as the means* to get the resulting set of nodes. sNREs enrich NREs in terms of output produced via our reconstruction algorithms (see Section 3) thus indirectly allowing to ask a larger class of requests. As an example with sNREs one can ask to retrieve *co-directors* and get as by-product *movies co-directed*.

Languages like ERPQs [4] or ρ-queries [1] are clearly more expressive than sNREs as they allow to manipulate paths via e.g., regular relations. Nevertheless this higher expressiveness comes at the cost of a higher complexity for query evaluation. Our goal in this paper is to focus on providing a structural characterization of low-complexity languages like NREs and showing that one can get more (i.e., structural information) with no additional computational cost.

2.2 Structural Graphs

Structural Graphs are used to formally capture the output of the evaluation of a sNRE and are the building blocks of the novel sNREs query semantics.

Definition 1 (Structural Graph). *Given a graph G, a sNRE expression ex and a set of starting node $S \subseteq nodes(G)$, a structural graph is a quadruple $\Gamma=(V, E, S, T)$ where $V \subseteq nodes(G)$, $E \subseteq triples(G)$ and $T \subseteq V$ is a set of ending nodes, that is, nodes reachable from nodes in S via paths satisfying ex.*

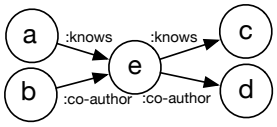

Fig. 1. An example graph.

Consider the graph G in Fig. 1 and the sNRE η=`(:knows/:knows)|(:co-author/:co-author)`. The answer with the semantics based on pairs of nodes (i.e., NREs semantics) is the set of pairs of nodes: (a,c), (b,d). Under the sNREs semantics, since there are two starting nodes a and b from which the evaluation produces results, one possibility would be to consider the Structural Graph (SG) capturing all results, that is, $\Gamma=(nodes(G), triple(G), \{a,b\}, \{c,d\})$. However, one may note that there exists a path (via the node e) from a to d in the SG even if the pair (a,d) does not belong to the answer. This could lead to misinterpretation of the query results and their structural information. To avoid these situations, we define G-sound and G-complete SGs.

Definition 2 **(G-Soundness)**. *Given a graph G and a sNRE expression ex, a SG is G-sound if, and only if, all ending nodes are reachable from all the starting nodes via some paths satisfying ex.*

Definition 3 **(G-Completeness)**. *Given a graph G and a sNRE expression ex, an SG is G-complete if, and only if, all nodes in a graph G reachable from some starting nodes, via some paths satisfying ex, are in the ending nodes.*

The SG Γ in the above example violates G-soundness because the only path existing (via the node e) from a to d (and from b to c) in Γ does not satisfy the expression η. The following lemma guarantees G-soundness and G-completeness.

Lemma 4 **(G-Sound and G-Complete SGs)**. *Structural Graphs having a single starting node $v \in nodes(G)$ are G-sound and G-complete.*

We are now ready to define answer graphs that capture the output of sNREs.

Definition 5 (**Answer Graph**). *Given a sNRE expression ex and a graph G, an answer graph $\mathcal{E}$ is a set of G-sound and G-complete structural graphs.*

The answer graph of our previous example is the set $\mathcal{E}=\{\Gamma_a, \Gamma_b\}$ s.t.:

1. $\Gamma_a=(\{a,e,c\}, \{\langle a, \texttt{:knows}, e\rangle, \langle e, \texttt{:knows}, \texttt{c}\rangle\}, a, \{c\})$
2. $\Gamma_b=(\{b,e,d\}, \{\langle a, \texttt{:co-author}, e\rangle, \langle e, \texttt{:co-author}, c\rangle\}, b, \{d\})$.

Note that $\mathcal{E}$ is an answer graph according to both $full$ and $filt$ semantics since there are no edges that do not contribute to reach some result node.

Formal Semantics. We devised two different evaluation semantics for sNREs that match the syntactic constructs $filt$ and $full$. In particular, the $full$ semantics considers all the parts of G that were visited during the evaluation of an expression, while the second one $filt$, only the portion of G that actually contributed to build the answer; in other words, the set of SGs such that each Γ_v only considers paths that starts from $v \in nodes(G)$ and satisfy the expression (i.e., union of all the successful paths). For sake of space we do not report the semantics that are available in an extended version of this paper [9].

3 Algorithms and Complexity

We now describe algorithms for the evaluation of sNREs expressions under the novel semantics. The interesting result is that the evaluation of a sNRE expression e in this new setting can be done efficiently. Let e be a sNRE expression and G a graph. Let $|e|$ be the size of e, Σ_e the set of edge labels appearing in it, and $|G|=|nodes(G)| + |triples(G)|$ be the size of G. Algorithms that build answer graphs according to the $full$ or $filt$ semantics are automata-based and work in two steps. The first step is shared and leverages *product automata*; the second step requires a *marking phase* only for the $filt$ semantics and is needed to include nodes and edges that are relevant for the answer.

Building Product Automata. The idea is to associate to e (and to each $[exp]$) a non deterministic finite state automaton with ϵ transitions $\mathcal{A}_e$ ($\mathcal{A}^{exp}$, resp.). Such automata can be built according to the standard Thomson construction rules over the alphabet $Voc(e)=\Sigma_e \cup \bigcup_{[e_1]\in e}$, that is, by considering also $[e_1]$ in e as basic symbols. The product automaton is a tuple $G \times \mathcal{A}_e = \langle Q^e, Voc(e), \delta^e, Q_0^e, F^e \rangle$ where Q^e is a set of states, δ^e:$Q^e \times (Voc(e) \cup \epsilon) \rightarrow 2^{Q^e}$ is the transition function, $Q_0^e \subseteq Q^e$ is the set of initial states, and $F^e \subseteq Q^e$ is the set of final states. The building of the product automaton $G \times \mathcal{A}_e$ is based on the algorithm used by [11] based on the labeling of the nodes of G wrt nested subexpressions in e.

Building Answer Graphs. We now discuss algorithms that leverage product automata (of the sNRE expression e and all nested subexpressions) to produce answer graphs according to the $full$ and $filt$ semantics. To access the elements of a structural graph Γ (see Definition 1) we use the notation $\Gamma.x$, with $x \in \{V, E, S, T\}$. The main algorithm is Algorithm 1, which receives the sNRE expression and the type of answer graph to be built. In case of the $filt$ semantics the data structure `reached`, which maintains a set of states (n_i, q_j), is initialized via the procedure `mark` (line 3) reported in Algorithm 2; otherwise, it is initialized as the union of: *(i)* all the states of the product automaton $G \times \mathcal{A}_e$; *(ii)* all the states of the product automata $(G \times \mathcal{A}_{exp})$ of all the nested expressions in e (line 5). The procedure `mark` fills the set `reached` with all the states in all the product automata that contribute to obtain an answer; these are the states in a path from an initial state to a final state in the product automata. As shown in Algorithm 2, `reached` is populated by navigating the product automata backward from the final states to the initial ones.

Input : sNRE e, graph G, output (*full* or *filt*)
Output: $\mathcal{E}$: an answer graph as set of SGs Γ_s

1: build the product automaton $G \times \mathcal{A}_e$
2: **if** *filt* /* filtered semantics */ **then**
3: `reached` = **mark**$(G \times \mathcal{A}_e, \emptyset)$
/* `reached` keeps nodes in $G \times \mathcal{A}_e$ that are in a path to a final state */
4: **else**
5: `reached` $= Q^e \cup \bigcup_{[exp] \text{ in } e} Q^{exp}$
6: **for all** $(s, q_o) \in Q_0^e$ **do**
7: $\Gamma_s = \langle \{s\}, \emptyset, s, \emptyset \rangle$
8: $\texttt{seen}_{(s,q_o)} = \{s\}$
/* `seen` for each state s_j keeps nodes in $G \times \mathcal{A}_e$ from which it has been reached */
9: $\mathcal{E} = \bigcup_{(s,q_o) \in Q_0^e} \{\Gamma_s\}$
10: `visit` $= \bigcup_{(s,q_o) \in Q_0^e} \{((s, q_0), \{s\})\}$
/* `visit` keeps nodes to be visited */
11: **buildA**$(G \times \mathcal{A}_e, \texttt{reached}, \mathcal{E}, \texttt{visit})$

Algorithm 1: BuildAnsG

Input: product automaton $G \times \mathcal{A}$, set of states `reached`
Build: set of states `reached`

1 `reached` $=$ `reached` $\cup \bigcup_{(n,q_f) \in F^e} \{(n, q_f)\}$
2 `visit` $= \bigcup_{(n,q_f) \in F^e} \{(n, q_f)\}$ s.t. $q_f \in F$
3 `visitN` $= \emptyset$
4 **while** `visit` $\neq \emptyset$ **do**
5 **for all** (n, q) in `visit` **do**
6 **for all** transition $\delta((n', q'), x) \in G \times \mathcal{A}^e$ s.t. $(n, q) \in \delta((n', q'), x)$ **do**
7 **if** $(n', q') \notin$ `reached` **then**
8 `visitN` $=$ `visitN` $\cup \{(n', q')\}$
9 `reached` $=$ `reached` $\cup \{(n', q')\}$
10 **for all** $[exp]$ in x **do**
11 **mark**$(G \times \mathcal{A}^{exp}, \texttt{reached})$
12 `visit` $=$ `visitN`
13 `visitN` $= \emptyset$
14 **return** `reached`

Algorithm 2: mark$(G \times \mathcal{A}^e, \texttt{reached})$

Then, the set of SGs composing an answer graph $\mathcal{E}$ are initialized (lines 6-7; 9) by adding to $\mathcal{E}$ an SG Γ_s for each initial state (s, q_0) of $G \times \mathcal{A}_e$. Moreover, the data structure `seen` is also initialized (line 8) by associating to each state (s, q_0) the node s (associated to the initial state (s, q_0)) from which it has been visited. Hence, `seen` maintains for each state, reached during the visit of the product automata, the starting nodes from which this state has already been visited.

The usage of `seen` avoids to visit the same state more than once for each starting node. The data structure `visit` is also initialized with the initial states of $G \times \mathcal{A}_e$ (line 10); it contains all the states to be visited in the subsequent step plus the set of starting nodes for which these states have to be visited. Then, the SGs are built via `buildA` (Algorithm 3). All the states in `visit` are considered (line 2) only once for the set $B_{n,q}$, which keeps starting nodes for which states in `visit` have to be processed (line 3). Then, for each state $(n, q) \in$`visit` all its transitions are considered (line 7); in particular, for each state $(n', q') \in$`reached`, reachable from some $(n, q) \in$ `visit` via some transitions (line 8), the set of "new" starting nodes (D) for which (n', q') has to be visited in the subsequent step is computed with a possible update of the sets `visit` and `seen` (lines 9-12).

Input: product automaton $G \times \mathcal{A}^{\bar{e}}$, set of states `reached`, Answer Graph $\mathcal{E}$, list of states to visit `visit`

```
1  visitN = ∅
2  for all (n, q) in visit do
3      B_{n,q} = ⋃_{((n,q),S)∈visit} S
4      for all s ∈ B_{n,q} do
5          if q ∈ F^ē then
6              add n to Γ_s.T
7      for all transition δ^ē((n, q), x) do
8          for all (n', q') ∈ δ^ē((n, q), x) s.t (n', q') ∈ reached do
9              D = B_{n,q} \ seen_{(n,q)}
10             if D ≠ ∅ then
11                 visitN = visitN ∪ {((n', q'), D)}
12                 seen_{(n',q')} = seen_{(n',q')} ∪ D
13             if x ∈ Σ_e then
14                 for all s ∈ B_{n,q} do
15                     add n' to Γ_s.V
16                     add (n, x, n') to Γ_s
17             else if x is a [exp] then
18                 let (n, q_0) ∈ Q_0^{exp}
19                 seen_{(n,q_0)} = seen_{(n,q_0)} ∪ B_{n,q}
20                 buildA(G × A^{exp}, reached, E, {(n, q), B_{n,q}})
21 buildA(G × A^ē, reached, E, visitN)
```

Algorithm 3: buildA($G \times \mathcal{A}$, `reached`, $\mathcal{E}$, `visit`)

If the transition is labeled with a predicate symbol in G (line 13), the SGs corresponding to nodes $s \in B_{n,q}$ are populated by adding the corresponding nodes and edges (lines 14-16). Otherwise, if the transition is a nested expression the building of the answer graph $\mathcal{E}$ proceeds recursively by visiting the product automata associated to all nested (sub)expressions.

Theorem 6. *Given a graph G and a sNRE expression e, the answer graph $\mathcal{E}$ (according to both semantics) can be computed in time $\mathcal{O}(|nodes(G)| \times |G| \times |e|)$.*

Proof (Sketch). The answer graph built according to the *full* semantics can be constructed by visiting $G \times \mathcal{A}_e$ (Algorithm 3). In particular, for each starting state (n, q), the states and transitions of $G \times \mathcal{A}_e$ are all visited at most once (and the same also holds for the automata corresponding to the nested expressions of e). The starting and ending nodes of each answer graph are set during the visit of the product automaton. For each node s corresponding to a starting state $(s, q_o) \in Q_0^e$ a structural graph is created (Algorithm 1, lines 6-7); the set of nodes reachable from s is set to be $\Gamma_s.T = \{\mathtt{n} \mid (\mathtt{n}, q) \in F^e$ and (n, q) is reachable from $(s, q_o)\}$ (Algorithm 3 lines 5-6). Thus, each structural graph can be computed by visiting each transition and each node exactly once with a cost $O(|Q^e| + \sum_{[exp] \in e} |Q^{exp}| + |\delta^e| + \sum_{[exp] \in e} |\delta^{exp}|) = O(|G| \times |e|)$. Since the number of SGs to be constructed is bound by $|nodes(G)|$, the total cost of building the answer graph $\mathcal{E}$, is $\mathcal{O}(|nodes(G)| \times |G| \times |e|)$. This bounds also take into account the cost of building product automata. In the case of the *filt* semantics, the marking phase does not increase the complexity bound; this is because the set `reachable`, which keeps reachable states, is built by visiting at most once all nodes and transitions in all the product automata, with a cost $O(|G| \times |e|)$. □

Note that in Algorithm 3, the amortized processing time per node is lower than $|G| \times |e|$ when visiting the product automaton since the Breadth First

Search(es) from each starting state are concurrently run according to the algorithm in [12]. Finally, the SGs in the $\mathcal{E}$ built via Algorithm 1 are both G-sound and G-complete. It is easy to see by the definition of the product automaton, that there exists a starting state (n, q_0) that is connected to a final state (n', q_f) in $G \times \mathcal{A}_e$ and, thus, a path from n to n' in Γ_n iff, there exists a path connecting n to n' in G satisfying e.

4 Concluding Remarks and Future Work

We have studied the novel problem of characterizing graph languages from the point of view of the structure of the graph where queries are evaluated. We picked the language of NREs as yardstick because of its attractive computational properties and showed how we can get more (in terms of structural information) at the same computational cost. We had to face challenges in terms of formalization of novel semantics that can deal with graphs and non-trivial (automata-based) algorithms that can capture structural information in an efficient way. In an extended version of this work [9] we discuss an experimental evaluation and an instantiation of the framework to tackle the problem of explaining graph queries. This works opens some challenging research questions such as: (i) consider a more expressive navigational core [7] and enhance it with structural information; (ii) add negative information (e.g., parts of a query that failed) (iii) provide structural information according to user-specified preferences [8].

References

1. K. Anyanwu and A. Sheth. p-Queries: Enabling Querying for Semantic Associations on the Semantic Web. In *WWW*, pages 690–699. ACM, 2003.
2. M. Arenas, S. Conca, and J. Pérez. Counting Beyond a Yottabyte, or how SPARQL 1.1 Property Paths will Prevent Adoption of the Standard. In *WWW*, pages 629–638, 2012.
3. M. Arenas, G. Gottlob, and A. Pieris. Expressive Languages for Querying the Semantic Web. In *PODS*, 2014.
4. P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive Languages for Path Queries over Graph-Structured Data. *ACM TODS*, 37(4):31, 2012.
5. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of Conjunctive Regular Path Queries with Inverse. In *KR*, pages 176–185, 2000.
6. M. P. Consens, J. W. S. Liu, and F. Rizzolo. Xplainer: Visual explanations of xpath queries. In *ICDE*, pages 636–645. IEEE, 2007.
7. V. Fionda, Pirrò G., and M. P. Consens. Extended Property Paths: Writing More SPARQL Queries in a Succinct Way. In *AAAI*, 2015.
8. V. Fionda and G. Pirrò. Querying Graphs with Preferences. In *CIKM*, pages 929–938. ACM, 2013.
9. V. Fionda and G. Pirrò. Explaining Graph Navigational Queries. In *ESWC*, 2017.
10. L. Libkin, J. Reutter, and D. Vrgoč. Trial for RDF: adapting graph query languages for RDF data. In *PODS*, pages 201–212, 2013.
11. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. *Journal of Web Semantics*, 8(4), 2010.
12. M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *VLDB Endowment*, 8(4):449–460, 2014.